Logic and Lattices for a Statistics Advisor

Richard A. O'Keefe

Ph.D.
University of Edinburgh
1985

Table of Contents

I declare that this thesis has been composed by myself, and that the work presented is my own, with the exception of the focussing algorithm described in Chapter 5, the description of which is due to Gordon Plotkin.

Abstract

The work partially reported here concerned the development of a prototype Expert System for giving advice about Statistics experiments, called ASA, and an inference engine to support ASA, called ABASE.

This involved discovering what knowledge was necessary for performing the task at a satisfactory level of competence, working out how to represent this knowledge in a computer, and how to process the representations efficiently.

Two areas of Statistical knowledge are described in detail: the classification of measurements and statistical variables, and the structure of elementary statistical experiments. A knowledge representation system based on lattices is proposed, and it is shown that such representations are learnable by computer programs, and lend themselves to particularly efficient implementation.

ABASE was influenced by MBASE, the inference engine of MECHO [Bundy *et al* 79a]. Both are theorem provers working on typed function-free Horn clauses, with controlled creation of new entities. Their type systems and proof procedures are radically different, though, and ABASE is "conversational" while MBASE is not.

Chapter 1
Background

1.1. Lies, Damned Lies, and ...

> **sta-tis-tics** *n.* **1.** (*functioning as sing.*) a science concerned with the collection, classification, and interpretation of quantitative data and with the application of probability theory to the analysis and estimation of population parameters. **2** the quantitative data themselves. [C18 (originally "science dealing with facts of a state"): via German *statistik* from New Latin *statisticus* concerning state affairs, from Latin *status* STATE]

1.1.1. Why Bother?

The goal of my research was to produce an Expert System for Statistics consulting. The program, called ASA (Automatic Statistics Assistant), asks its client questions about the structure of his experiment, the nature of his measurements, and what he wants to learn from the experiment, and then suggests how the data from such an experiment may be analysed.

While I was studying for my BSc and MSc, I did a lot of consulting at the University Computer Centre. I was appalled to see how often postgraduate students in the human sciences (psychology, education, medicine, anthropology, and so on) would simply leaf through the manual of a statistical package until they came to something their data could be put into, and were happy with whatever numbers the magical computer blessed them with. Worse still, gross mistakes in experimental design went undetected, not just by the students, but by their supervisors too. There is no reason to think that University is particularly bad. When I came to Edinburgh I found the same problems all over again.

Nor is the problem confined to universities. It is said (I cannot recall the reference) that 70% of "statistics" in published medical research is of dubious merit.

1.1.2. What is Statistics, anyway?

The heading of this section refers to the "facts of a state" mentioned in this entry from the Collins English Dictionary. Said facts still form part of Statistics, a part known as Sample Survey Theory [Cochran 77], and despite the best efforts of able statisticians retain their

potential for deception. Indeed, statistical analysis is fraught with pitfalls for the unwary, and interpretation is beset with dangers. Why is this so?

A definition of Statistics which some statisticians prefer is that it is the science of reasoning about inexact data. But Statistics is even more than that. Statistics is just the scientific method in action. It is about forming beliefs from data, deciding how to investigate beliefs, designing experiments to test these beliefs or to fill in partially specified models, and about how you revise your beliefs in the light of results. Of course, there is more to science than this, as [Kuhn 70] has argued. Statistics is only applicable to "normal science". In order to design an experiment you need to have a stable paradigm.

Different parts of the scientific process are served by different areas of Statistics. For argument's sake, let us pretend that the following "algorithm" captures part of the "normal science" process.

```
[1]  Decide what phenomena to study.
[2]  Collect some data relating to these phenomena.
[3]  Look for interesting patterns in the data.
[4]  Repeat steps 2 and 3 until something emerges.
[5]  Formulate a model incorporating the patterns.
     If possible, design a model with explanatory as
     well as predictive power.
[6]  Design an experiment to test this model.
[7]  Perform the experiment.
[8]  Analyse the data from the experiment.
[9]  Repeat steps 6 to 8 until the model fails.
[10] Use the collected data to refine the model, or
     to suggest a replacement for it.
```

Step 1 is necessarily a human activity. Whether to study racial differences, the effectiveness of psywar techniques, or the improvement of crops is a moral question rather than a statistical one. Step 2 is pre-theoretic, and would be very difficult to assist. An important methodological question for research at this stage is whether some datum forms part of the subject or not. E.g. do reports of stones falling from the sky belong in a study of meteorology? However, people are never really in a state of complete ignorance. They usually have some theory (however mistaken) to guide their search, so that some of the principles relating to step 6 can be used. When we come to step 3, though, there is a body of known techniques which can be used. This is the area of Statistics called **Exploratory Data Analysis** (EDA). It is a new area, and one of great interest. Humans are very good at detecting certain kinds of patterns, and very bad at detecting others. EDA is a collection of hints about what kinds of pattern to look for, and a body of methods for displaying data to help you see them.

Step 5 is largely unexplored. [Langley *et al* 83] have done some work on theory formation, but they have worked with very restricted "data". Rules of thumb have been

published, but they tend to boil down to (1) if it's like something else that already has a model, try to adapt that one, and (2) try to form models which mean something. Dimensional analysis is one of our most powerful guides here. [Peters 83] is a fascinating account of power law models in biology which illustrates this problem.

Step 6 is where the greatest need for statistical consultants arises. There is a body of Statistics theory called **Design of Experiments**, which is not actually a general theory of experiments, but is concerned with structural patterns in a useful subclass. It is an outgrowth of early work in Agricultural research, and still retains a vocabulary derived from fields and crops. It is applicable to other problem areas, such as determining the optimal yield point of a chemical process which is too complex to use exact chemical models. There is another body of theory called **Sample Survey Theory** [Cochran 77] which is concerned with how to select a sample (typically for a poll or census or similar survey, hence the name) so as to get the most favourable cost-precision trade-off. There is also a collection of anecdotal material and broad hints about such things as questionnaire design, possible sources of bias {a statistical term relating to systematic error, not to be confused with prejudice}, and usually a repertoire of "standard" experiments in the researcher's domain. Alas, designing an effective and affordable experiment requires a great deal of knowledge, much of it real world knowledge. I am confident that a major AI research project could produce a usable program working in a limited domain. My research is not directed to this problem, but should be relevant.

Step 8 is where most of the statistical work has been done. There are methods for testing nearly everything imaginable, and new methods to meet actual needs are developed nearly every month. The mathematics of this area, called **Confirmatory Data Analysis**, is very well developed, and is constantly being expanded. This is one reason why an automatic consultant such as ASA can be useful: today's best method for some problem may be superseded tomorrow, and the client cannot be expected to keep up with the latest developments. This is to view ASA as a polyalgorithm, and for an experienced client that would be the only gain from using it. But for the naive client, a much more important result of using a computer program is that it will not forget to ask the right silly questions.

The work reported here is directed at steps [3] and [8]. This involves an elementary attack on [5], in that fitting a statistical model often involves re-expressing the data (perhaps taking square roots, or logarithms), which is a very basic sort of model design. The assumptions built into the structure of ASA are that

- the client knows a fair bit about what the entities he is studying are like, even if he does not know all the laws and relationships governing them. He must, in particular, be able to classify them.
- the client knows something about his "measuring instruments", so that ASA can work out what kinds of statistical variables they are.

- the client has either performed his experiment or has a definite experimental plan in mind, and knows what happened when to which entities, or what he intends to happen. ASA can sometimes manage with a vague idea of the structure of an experiment, but it is assumed that any question it might pose about the structure has a firm answer decided before the consultation.

- it is the client's job to decide what is "interesting", not ASA's.

In particular, I make no pretence of tackling step [6].

1.1.3. Hair of the Dog.

I have a friend who is intelligent and capable, a highly skilled commercial programmer with an MSc in Operations Research. He had a lot of difficulty with Statistics, and has never really understood why you have to use different confidence interval for testing a hypothesis you thought of after seeing the data from the interval you use to test a hypothesis you were interested in before. For example, suppose you are comparing two brands of paint, and want to estimate how much more of your house you can cover with a can of brand A. Surprisingly enough, you get a much sharper estimate if you predicted beforehand that brand A was the more economical than if you noticed this afterwards. How can looking at the results make such a difference? Yet you get different precisions because different variables are being estimated. This is one of the easiest traps to explain: if you thought beforehand that A was better than B, you are estimating the difference between A and B. But if you noticed after the experiment that A was better, you are estimating the difference between the better and the worse paint. If you plan the analysis before you do the experiment, this distinction will not escape you. But if you do the experiment and then try to analyse it, the distinction is very often missed. If someone like my friend has difficulty with this concept, how much more will someone who has little Mathematics and no Statistics? ASA distinguishes between the goals "check which treatment is better and test whether this is reliable" and "check whether this treatment is definitely better than that". It cannot ensure that the client will give the right answer to the question, but at least it can ensure that the question will not be forgotten.

There are many computer programs available for doing statistical calculations. The trouble is that most of them are designed for experts: each package "trusts" its users to ensure that the analyses they ask for make sense. The packages themselves have no "understanding" of Statistics, let alone of the user's subject area; all they are good at is data management and complicated numerical calculations.

The problem which motivated the design of ASA, then, is that automatic computers make it possible for people who do not understand what they are doing to perform statistical calculations and come away with the false belief that the numbers mean something.

Other researchers have seen the problem. For as long as statistical packages have existed, statisticians have been urging that they should check their answers. For example, if a regression line is strongly influenced by a single data point, a package should warn its user. Statisticians have developed several of these "diagnostics" (extra calculations whose results can indicate problems with the analysis) and continue to develop new ones. Diagnostic tests can be added to a statistical package without changing the way that packages are programmed or used. AI techniques are not needed for that.

But diagnostics only warn you of problems in the data, and even packages which make all sorts of checks still rely on their user to be doing the right *kind* of analysis. But that is precisely where naive users make their most serious mistakes.

My research, then, was

- **To discover what kinds of knowledge**
- **and what kinds of reasoning are needed to work out *valid* analyses for *simple* statistical experiments,**
- **to determine what AI techniques are needed if a computer program is to perform this task,**
- **and to explore techniques for efficient logic-based expert systems generally.**

1.2. Even Textbooks Make Mistakes

One way of testing the classification of measurements developed in chapter 2 and the structure of experiments developed in chapter 3, is to take examples from Statistics textbooks and see whether ASA recommends the same method of analysis as the textbook. Generally, either ASA does recommend the same method, or else it fails to recommend anything. But when it recommands a different method, it is usually the case that ASA is right and the book is wrong. Of course this requires an independent way of telling which is right. Initially I relied on my own judgement, but a statistician at the University of Auckland and a statistician at the Department of Scientific and Industrial Research to whom I showed the results agreed with me and with ASA. (Both of them are practising statisticians with PhDs.)

1.2.1. Inveterate Liars

Here is an example taken from page 52 of [Everitt 77]. Since Everitt gives very little of the experimental structure, any assistance that mechanical methods could give us must be derived entirely from the types of variables rather than how we got them.

A sample of 223 boys were classified by age and by whether or not they were inveterate liars. The table he presents is this

Age:	5-7	8-9	10-11	12-13	14-15	Total
Liars	6	18	19	27	25	95
Non-liars	15	31	31	32	19	128
	21	49	50	59	44	223

The task is to find a model for the proportion of liars in terms of age. ASA is able to solve this problem.

A standard technique for fitting one continuous variable as a function of another is linear regression. But it never makes sense to present the client with a model of the form

```
proportion = a*continuous + b
```

as there will almost always be reasonable values for the continuous variable for which the proportion takes on values outside the range [0,1]. The standard transformation for proportions is *logits*. In "Exploratory Data Analysis", J.W.Tukey says "... the writer usually tries flogs [folded logs] even before percents (plain or folded) or pluralities -- flogs are the first-aid bandages for counted fractions." The difference is that if the proportion is measured as M out of N,

```
logit(M/N) = log(M/(N-M))

flog(M,N)  = 0.5 log((M + 1/6)/(N-M + 1/6))
```

The difference between folded logs (flogs) and logits is a factor of two and the additional 1/6 counts to cope with zeros. They are basically the same thing. We can interpret a logit or flog as the logarithm of the *odds* on an event rather than the *probability*. logit ranges from -infinity to +infinity and is symmetric about proportion 1/2.

So one model is

```
logit(proportion) = a*continuous+b                    [0]
 <=> proportion = 1/(1 + b1*a1^(-continuous/scale))
```

A statistician friend told me that this would be her automatic reaction. Logits have the advantage that equation [0] makes sense for all values of the continuous variable, both positive and negative. ASA knows about the logit transform, but not about flogs.

If the continuous variable is strictly positive, as in this case, we may be able to produce a model where the proportion cannot be negative, or cannot exceed one, but the other limit may still be exceeded. There are two possible transformations which are not too hard to interpret. They are

```
log(proportion) = a*continuous+b                      [1]
 <=> proportion = b1*a1^(continuous/scale)
```

```
log(1-proportion) = a*continuous+b                [2]
 <=> proportion = 1 - b1*a1^(continuous/scale)
```

where scale is a constant to produce a dimensionless power.

Clearly b1 and a1 must be positive, but that imposes no restriction on the a and b coefficients. Now model [1] has the property that the proportion will approach zero but never attain it, but may fail because the proportion can exceed one. Model [2], on the other hand, can approach but never equal one, but may turn negative. If the explanatory continuous variable has a lower bound and "a" is negative, both of the constraints are satisfied. So to choose between models [1] and [2], we have to ask the client whether he believes that the proportion should decrease as the continuous variable increases (pick [1]) or decreases (pick [2]). In this case, the proportion of liars is expected to increase, so the model to pick is model [2]:

```
log(proportion(non-liars)) = a*age + b
```

where a < 0

In this case, [2] actually makes more sense than [0]. What we have is the familiar logistic curve of population growth up to a limit. Starting at the time he learns to talk (when log(1-p) = 0), there is a fixed probability per unit time that a boy will become a liar. "a" measures that probability, and "b/a" measures the time of learning to talk.

Everitt now goes on to fit a linear regression. The model I obtained from GLIM by doing this was

```
(proportion of liars) = 0.047 + 0.033*(age in years)
```

This model fits the observed data very well; the predicted numbers of liars in each group is only out by 1. However, it has two deficiencies. First, it claims that 5% of newborn infants are inveterate liars, something I find difficult to believe. How many babies say "Da Da" with intent to deceive? Second, it predicts that *more* than 100% of people aged 29 or older are inveterate liars. (Could this be the otherwise unobtainable justification for "Never trust anyone over 30"?) The customary reply to this from analysts who do linear regressions on raw data is "We're never going to apply the model outside its range of validity". To which I reply, "You mayn't, but somebody else is bound to". For all too often, the model is published without indicating what range of data it is based on.

What happens if you fit model [2]? GLIM came up with

```
{proportion of liars} =
    1 - 1.101*0.943^{age in years}
```

Now this model appears to predict that -10% of infants are inveterate liars, which isn't much better than +5%. However, if we look more carefully, we see that there is an interesting figure in there. The age at which the proportion of liars is zero is 1.6 years, which is a plausible sort of estimate of when children might start talking. We can interpret this model very easily, as "The proportion of liars at ages below 2 years doesn't make much sense, as

too few of them can talk yet. Otherwise, 5.7% of truthful children become inveterate liars each year." This model has a definite lower bound, which reflects a genuine lower bound in the age variable, but it has no upper bound. It predicts that 80% of 29-year-olds are inveterate liars. I have less difficulty believing this than 120% !

How well does the meaningful model perform? Oddly enough, it doesn't do as well as Everitt's one. For ages 10-11, model [2] predicts 29 liars, Everitt's predicts 29.5, and there were 31. But it is never more than 1/2 a unit worse than Everitt's model. This is a small price to pay for having a wider range of validity.

1.2.2. A Knotty Problem.

The second example comes from [Siegel 56]. The experiment was intended to illustrate a psychological phenomenon called "regression under stress". A sample of about 30 students was divided at random into two groups. All of the students were taught two methods (call them A and B) for tying a certain kind of knot, but students in the first group were taught method A first and then method B, while students in the second group were taught method B first, and then method A. Shortly after this, the students were given a four hour examination. After the exam, each student was asked to tie the knot, and the method used was recorded.

Anticipating the notation presented in Chapter 3[1], the structure of this experiment is

```
select[first_taught;
        A -> teach(A) then teach(B),
        B -> teach(B) then teach(A)]
then
measure[method_used]
```

where first_taught and method_used take values {A,B}. Since first_taught is a treatment the experimenter chose, there is no point in looking for an explanation of it. Since method_used is a measurement which the experimenter did not control, there is some point in studying it, and since first_taught happens before method_used is measured, it should be in the explanation. This is in fact the way ASA works: first consider which variable might need explaining, and then check which variables should appear in the explanation. So just looking at the structure of the experiment tells us that a good thing to do is

```
look for an explanation of method_used
in terms of first_taught
```

Since both variables take only two values, the results of the experiment can be reported in a two-by-two table:

---

[1]Ignore the punctuation; just read the words.

first_\ taught \	method_used A	B
A	Naa	Nab
B	Nba	Nbb

where Nij is the number of subjects who were taught method i first and used method j when they were tested.

Simply deciding that this is the appropriate way to summarise the results of the experiment is a significant achievement in itself. But two-by-two tables are so common that a statistician is likely to know a lot about them. To start with, there are at least five ways that this table could be "simple":

- method_used could tend to be the same as first_taught (Naa and Nbb large, Nba and Nab small); this is the one we expect
- method_used could tend to be the opposite of first_taught (Nab and Nba large, Naa and Nbb small)
- method_used could tend to be mostly A (Naa and Nba large, Nab and Nbb small)
- method_used could tend to be mostly B (Nbb and Nab large, Nba and Naa small)
- method_used could be completely random (all counts much the same)

There are well known methods for checking each of these possibilities.

However, Siegel doesn't do this. In his description of the experiment, all that is recorded is one number for each student: whether he used the method he was taught first (1) or the method he was taught second (2). "Regression under stress" suggests that most students should use the method they were first taught, so Siegel tests whether the 1s significantly outnumber the 2s.

There are several points to make about this. One is that the test Siegel uses is less sensitive; if there is a weak tendency to use the first method taught, testing for a predominance of 1s may fail to detect it, while examining the 2-by-2 table may reveal it. Another is that if method A is overwhelmingly simpler than method B, it will be apparent in the 2-by-2 table that most students are using method A, while in Siegel's method it will just appear as "no effect".

But the main point is that the *better* analysis was *easier* to find than the worse. Using the 2-by-2 table comes directly from the fact that the students are first divided into 2 groups and then a measurement with 2 possible outcomes is made. To obtain Siegel's analysis, you have to combine first_taught and method_used into one variable.

1.2.3. Summary

These two examples showed that elementary reasoning about such questions as

- what sort of measurement is this?
- who decided what the value of this measurement would be?
- did this happen before or after that?

could lead to valid analyses. These two examples were chosen because the textbooks they came from presented inferior analyses. Other examples could have been chosen: one textbook on regression even has an example where logarithms of negative numbers are taken! The irony is that the rules which led to the better analyses were taken from the very textbooks that failed to apply them.

1.3. ASA

ASA's rule base is not described in this document. The reason for that may be a surprising one. Many authors have stated that "knowledge acquisition" is a bottle neck in the development of Expert Systems. Indeed, this is often used as the justification for induction programs. (Which I have also studied. See [O'Keefe 83a] and [O'Keefe 83b].) In the domain of analysing elementary statistical experiments, however, I found the exact opposite to be the case.

ASA has a collection of rules that describe statistical methods. They are taken directly from introductory textbooks. It has a small number of heuristic rules to guide model selection and transformation of variables. In the past such rules were not stated explicitly in Statistics textbooks, but even so it was not hard to discover them. With the advent of Exploratory Data Analysis, that practice is changing, and several of the heuristic rules I thought I had discovered are explicitly stated in [Mosteller 78]. When applied to the simple experiments which ASA is intended to be good at, these simple rules are astonishingly effective.

The planning algorithm that ASA uses is a straightforward one:

- pick a method which is relevant to the current goal
- check that the part of the experiment it refers to has the right structure
- ensure that the entities it refers to have the right types (setting up subgoals to create suitably transformed variables if necessary)
- set up the method's subgoals

This bears a strong family resemblance to the Marples algorithm [Marples 74] used by MECHO [Bundy *et al* 76a].

That part of the inference engine which deals with ordinary Horn clauses is based on David Warren's "Earley deduction" technique [Pereira & Warren 83].

If the inference engine, major control structure, and rules are either taken from existing work or easy to discover, what then was hard?

What made writing ASA hard was developing a notation in which experiments could be described and rules stated.

Chapters 2 and 3 describe the classification of measurements and the structure of experiments because that is the most interesting thing about ASA, and the most innovative. Three working statisticians with whom I have discussed this material have told me that they are in basic agreement with it (but haven't checked the details, and are not responsible for any of the mistakes), that bearing this formalism in mind would help them in their own work, and even that they have learned something from it.

1.4. Outline of Contents.

When reading this volume, bear in mind that there was never any intention of producing a program which could actually be used by naive clients. That would have involved massive man/machine interface problems. To repeat the end of section 1.1, the goal of my research was

- To discover what kinds of knowledge
- and what kinds of reasoning are needed to work out *valid* analyses for *simple* statistical experiments,
- to determine what AI techniques are needed if a computer program is to perform this task,
- and to explore techniques for efficient logic-based expert systems generally.

This volume concentrates on knowledge representation issues. A program, ASA, was implemented, which was able to find valid analyses for simple experiments. This involved a planning strategy, a methodology for describing statistical methods, and an inference engine, none of which are described here as they are not directly concerned with knowledge representation as such. Many of the algorithms described in this volume are not used in ASA because simpler brute-force methods were adequate for small problems. They have, however, been tested.

This work has two major divisions.

1.4.1. Statistics

This division is about Statistics. It contains two chapters. It describes some of the knowledge which went into the program ASA. The purpose of this division is to illustrate some of the problems which face a knowledge engineer working in this area, and to motivate the techniques described elsewhere in this volume. ASA uses other bodies of information as well. For example, there is a lattice of goal types, and there are descriptions of statistical methods. The techniques I developed to handle measurements and experiment structure are capable of handling them. In order to keep the text to a manageable size, I do not describe the goal types and method descriptions. The planning component of ASA is not described because the algorithm is essentially trivial: nothing beyond [Warren 74] and [Bundy *et al* 79b] was needed.

1.4.1.1. Chapter 2

Chapter 2 describes the structure of the Statistics world (not the structure of experiments) which should be useful for any sort of mathematical modelling. We have

- objects, classified by "descriptions"
- with components (other objects) and properties
- processes applied to objects and values produce measurements of properties

The approach is similar to frames, or to the Entity-Relation model in data base theory.

The bulk of the chapter describes the lattice of value spaces, the fundamental way that measurements and variables are classified.

1.4.1.2. Chapter 3

Chapter 3 describes the structure of experiments. The approach taken in the chapter is to treat an experiment as a process operating on streams of subjects, and to model this as one would model a programming language. Each particular part of the notation is motivated by an example showing a statistically meaningful distinction which must be made.

A practically important aspect of the notation is that if the substructure of part of an experiment turns out to be unimportant, it can be left vague. So ASA can start with an unspecified experiment, and refine it only as necessary.

1.4.2. AI

This division contains four chapters. It describes techniques which were developed for the "shell" which ASA runs under, called ABASE. (Pronounce this as "ay base".) The major theme of this division is that lattices are a good basis for descriptions. Other aspects of ABASE (such as the use of Earley deduction and annotations) are summarised in Chapter 8.

The point of this division is to answer the questions "How can I represent the kinds of knowledge described in the first division? How can I interpret these representations efficiently?"

1.4.2.1. Chapter 4

Chapter 4 is about "descriptions".

This chapter starts by pointing out some of the deficiencies of simple type systems, and of MECHO. "Descriptions" are developed from a very primitive basis: the idea that there are entities, there are descriptions, and there is a primitive relation "entity satisfies description" between them which is defined by the knowledge engineer. Sets of descriptions possessing some desirable properties turn out to be lattices.

I argue that "vague descriptions" may be a better approach to uncertain reasoning than "certainty factors", as they attribute imprecision to perceptions rather than predications.

I show that a form of negation is available using descriptions which does not rely on the closed world assumption, nor does it require an inference engine capable of handling anything but Horn clauses.

This form of negation can be used to provide a logically consistent approach to "defaults".

1.4.2.2. Chapter 5

This chapter presents a known learning algorithm, the "focussing algorithm". It is relevant for three reasons.

- It would be useful if the conditions for using methods could be learned. Apart from descriptions, ABASE uses function-free Horn clauses, and they can be learned from examples, so it is interesting to know that descriptions can be learned from examples as well.
- The fact that descriptions form a lattice was arrived at by considering specialisation. The focussing algorithm is concerned with generalisation. It is interesting that this route also leads to lattices.

- Chapter 4 introduces description-to-description rules, and shows that the closure of these rules again yields a lattice. This means that the focussing algorithm can handle background information in the form of description-to-description rules, which was not previously known.

1.4.2.3. Chapter 6

This chapter presents an algorithm for finding fixed points, and shows that its cost is proportional to the size of the problem. This means that description-to-description rules can be applied efficiently, provided the lattice of descriptions has finite depth. This approach is also applied to forward chaining in sets of function-free Horn clauses.

1.4.2.4. Chapter 7

Chapter 4 argues that a *single* type tree is not adequate, and proposes a lattice based approach with however many "aspects" the knowledge engineer finds convenient. While type trees are not capable of bearing the whole burden of description, they are excellent for some aspects, and ASA has several of them (object types, goal types, method types). It is therefore useful to handle type trees efficiently.

This chapter presents a dynamic node numbering method which makes it possible to test whether one node in a tree is an ancestor of another node in constant time. This can be used to perform some simple but useful inferences fast.

The method can also be combined with AVL trees or priority search trees to yield an efficient data structure for multiple context data bases.

1.4.3. End Matter

1.4.3.1. Chapter 8

Chapter 8 summarises the contributions of the work, and presents directions for future research.

In order to keep this volume to a reasonable size, and to avoid distraction from the main theme, several aspects of ABASE are not described in detail. These aspects include the use of Earley Deduction (due to D.H.D.Warren) as the basis of the inference engine, the treatment of negative information, the use of MECHO-like annotations on predicates, dependency maintenance, and "natural language" output generation. Chapter 8 summarises these aspects. Negative information and dependency maintenance are also touched on in the main body of the work.

1.4.3.2. Appendix A

Appendix A describes how experiment structures are actually handled in ASA.

1.4.3.3. Appendix B

While doing the research which resulted in chapter 2, I discovered a new kind of scale of measurement. Appendix B presents this scale, and a statistical method which uses such measurements as explanatory variables.

1.5. Notation

The notation and terminology I use when talking about lattices are standard mathematics. In particular,

```
X /\ Y  is the greatest lower bound

X \/ Y  is the least upper bound
```

I use **0** and **1** for the least and greatest elements of a lattice. (This comes from definition 1.8 in [Gierz 80].)

Some algorithms are written in the programming language C, and that code can be typed into a computer and compiled as it stands. Others are given in a sort of "pidgin" which is based on Algol 68 and on Dijkstra's notation. In particular,

```
X, Y, Z := e1, e2, e3

if Test then True else False fi

while Test do WhileTrue od
```

are the forms of (parallel) assignment, of conditionals, and of loops respectively.

Statements of ordinary logic are written using the connectives '&' (and), '|' (or), '⇒' (material implication), '⇔' (biconditional) and '~' (not), and the quantifiers '(∀x)' (for all) and '(∃y)' (there exists).

Other logical formulas appear in the text, using the connectives '&' (and), '->' (if-then), and 'not'. Another connective '-?->' is used which has no proper logical reading; the formula

```
Hypotheses -?-> Conclusion
```

is to be read as "if the Hypotheses are true, it may be worth assuming that the Conclusion is true as well, but be prepared to retract that assumption." Variables start with a capital letter, and are universally quantified unless introduced by the quantifier 'y^' (there exists). These are examples of the forms that ABASE interprets. The notation is to a large extent based on Prolog, especially the rather odd notation for existential quantification.

For typographic reasons, I use X =/= Y in text to state that X is not equal to Y.

The form "$X =_{def} Y$" is to be read as "X is defined to be Y".

1.6. Acknowledgements.

When I first came to Edinburgh I was encouraged to find the AI department full of pleasant, helpful people. I have never had any cause to revise this opinion.

I owe a special debt of gratitude to my supervisor, Alan Bundy, who has been patient with me, has repeatedly reassured me that my work was worth writing up, and has consistently encouraged me to get my thesis written instead of hacking around. He also advised me on the structure of this volume.

Lawrence Byrd was my co-supervisor while I was at Edinburgh. He explained MECHO to me, and help me put together the first version of ASA, which used MBASE as its base. His advice on how to construct an axiomatisation was helpful.

There are two people at Quintus without whom this volume might never have been finished: David Warren and Doug deGroot. Both of them found me the time to work on it and metaphorically "got behind me and pushed". I am particularly grateful to Doug deGroot: of several people who promised to read my draft and criticise it, he and Alan Bundy are the only people who actually did.

The British Council gave me a Commonwealth Scholarship, which supported me in Britain for 3 years. Computing was done on the Edinburgh ICF KL-10 under SERC grant GR/C/06226.

I owe my parents more than I can say. Such debts can never be paid back, only forward.

Chapter 2
Value Spaces

2.1. Introduction

This chapter describes part of ASA's basic representation structure. The next chapter describes how the structure of a statistical experiment is represented. This one is about objects, measurements, and statistical variables, and what we might want to say about them. It is important to realise that these two chapters are talking about *statistical knowledge*, about the kinds of statements that an Expert System in this domain should be able to represent. ASA also represents several things not discussed in this volume (such as goals and plans). There are still other things (such as measurement and treatment processes) which ASA does not represent, but which should be represented in a more capable program. Variable types and experiment structure seem to be the irreducible minimum for reasoning about experiments.

The point of value spaces in ASA is to have method of classification from which all the necessary statistical distinctions can be *derived*, yet which can be defined in terms that make sense to the client irrespective of whether he intends to perform statistical calculations on the results or not.

Chapter 4 develops a theory of descriptions which is based on the conventional AI notion of a type hierarchy. The approach presented there may seem unnecessarily complex, until you realise that it is needed to handle the approach to classification used in this chapter.

2.2. Objects, Properties, and Components

When we talk about statistical experiments, there are lots of things to give names to and reason about.

2.2.1. Objects

Statistical experiments are performed on material objects, or on related groups of material objects. For example, a quality control experiment to determine the state of a bolt manufacturing process will examine real bolts, or collections of bolts made at the same time. My interest in Expert Systems for Statistics was awakened by seeing how often "social scientists" misuse statistical packages; their experiments almost invariably work with human beings. Census data deal with households, that is, with groups of human beings logically related by having the same dwelling place.

It is not always the case that the objects which are studied have an independent existence. A medical experiment might involve taking several blood samples from a patient and doing different things to them. Before the samples were taken, they were all part of one continuous stuff, and it would be nonsense to suppose that a particular millilitre of blood could have been identified in advance. The distinction between "natural individuals" and "selections from a continuum" is important both physically and practically. However, for the elementary experiments and methods I have studied, it is acceptable to treat both kinds of objects the same. From now on, the unqualified word *object* will always mean *material* object, irrespective of whether the object in question is a natural individual or a selection. The word *entity* will include material objects and individual objects of thought.

ABASE makes the *unique name* assumption. That is, it assumes that every entity which must be considered in any particular experiment either possesses a name which the client can supply or may be assigned a unique name from which a description can be recovered which will enable the client to recognise the entity. A consequence of making this assumption is that ABASE can check whether two entities are the same or distinct by checking the names it uses for them. It is up to the knowledge engineer using ABASE to formulate his axioms so that this assumption is valid. For ASA this was no problem.

Objects belong to *object types*, which are arranged in an *object taxonomy*. For the elementary experiments and methods I have studied, it is sufficient for the object taxonomy to be a simple type tree (see Chapter 4 for a definition of this term), that is, a Linnaen style of classification is appropriate. An object taxonomy is useful because objects belonging to the same class (taxon) share many properties. They are said to *inherit* the shared properties from the class. For example, it is obviously useful if a statistical Expert System needs only to be told that the subjects of a particular experiment were dogs, and can then use general knowledge about the properties of carnivores, mammals, animals, living things, and material objects generally, to deduce that dogs can engage in a process calling running (of which speed and distance covered are properties with known characters), that they have an interesting body temperature (again, a property with known characters), that they have ages, weights, densities, and so on.

Note that dogs inherit the property "body temperature" from the class "mammal". They do *not* inherit the *value* of this property. Each mammalian species has its own characteristic body temperature. But different individuals, or the same individual at different times, may have different body temperatures. ABASE does not distinguish between attributes which are inherited and attributes which are not, but leaves it to the knowledge engineer to formulate his axioms so that the right inheritances are made.

2.2.2. Samples

A sample is a finite collection of objects (called *cases* or *sampling units*), all belonging to a common object type, and having to some extent a common history. An experiment may involve a single sample, or it may involve many samples of many kinds of objects. The cases which were actually studied are the *only* ones that an experiment can yield definite information about; extrapolation to other objects involves an act of faith. Chapter 3 describes samples more fully.

2.2.3. Populations

A population is a possibly infinite set of real or potential objects. We very commonly perform statistical experiments on small samples of objects in order to draw inferences about larger sets of the same kind of object. A large part of Statistics is concerned with how such inferences may be drawn and how unreliable the results are.

Establishing the validity of a generalisation is a subtle matter. Suppose we have a sample of 30 psychology students from Edinburgh University, and we measure their average "aggressiveness" according to some standard procedure. We can extrapolate this result to the whole population of psychology students at Edinburgh University with some confidence, provided we took care to take a random sample of students. But if the population we wanted to generalise to was all of the students at Edinburgh University, our estimates are likely to be biassed to an unknown amount in an unknown direction, and no amount of statistical reasoning or calculation will help. Only studying a more broadly based sample will help.

The elementary statistical methods built into ASA fall into three classes:

- those which refer only to the sample actually studied (such as the "exploratory data analysis" methods, which are basically tools for summarising the data)
- those which are valid inferences about the population *provided* the sample studied was a *random* subset of the population of interest
- those which only use the sample as a probe to study the treatments, and require the assignment of sampling units to treatments to be random

ASA takes the client's word concerning randomisation. Chapter 3 shows how we can represent the process of selecting the sample, that can sometimes reveal when the sample is not a random sample, and also discusses some varieties of randomised treatments.

Reasoning about populations can be quite subtle. When we have a random sample, one interesting population is the set of cases which *could* have been selected. Another population is the sets of cases we want to make inferences about. There are other distinctions which can be made as well.

The main points of interest about a population are

- what kinds of objects are in it
- how many objects are in it; if the population has several kinds of object, how many of each kind
- do the objects in the population exist regardless of whether we study them or not
- how do we *find* cases? Does the likelihood of our finding a case depend on its properties?

ASA represents populations only in the very weak sense that it asks the client what kind of object the sampling units are, and whether the sample constitutes the whole of the population of interest, or several other possibilities. It does not represent populations as explicit entities.

A full-scale Statistics Advisor should do so, especially if it were to help with census problems, which are notoriously plagued by sampling problems.

2.2.4. Properties and Components

Objects have (at least) two kinds of attributes. I distinguish between attributes whose values are other objects, which I call *components*, and attributes whose values are not objects, which I call *properties* (simply for the sake of having some name). More precisely, let T1 and T2 be object types, V be a value space (described later in this chapter, but think of it as something like "temperature" or "mass"), and let R1 and R2 be two binary relations.

If `R1 ⊆ T1 x T2,` **R1 is a** *component*.
If `R2 ⊆ T1 -> V,` **R2 is a** *property*.

This distinction is in principle a difficult one to draw: do numbers "exist" or do they just "describe"? There are two reasons why this is not a difficulty for ASA. The first is that it is not ASA's responsibility to decide. The second is that conventional kinds of measurement are always properties. A very important difference between components and properties is that it is possible to identify a component without error while it is not possible to identify properties without error. That is, for most practical purposes it does not make much dif-

ference how we determine which breeder a particular white mouse came from, we will get the same answer, and if we are wrong we are completely wrong, not just 5% out. While three different methods of measuring the animal's length could give three different answers (not converging no matter how often repeated) and yet agree quite closely.

When we provide a statistical expert system with "general knowledge", we want to be able to say things such as

```
every object has a weight

every mammal has legs

every vertebrate has a head
```

Since we also want to say that

```
variable w measures the weight of entity3

the typical_member of sample7 is
    the head of the typical_member of sample6
```

it would be natural to use second-order logic (as we want to use weight and so on as functions, and also to make statements about them as if they were individuals).

However, second-order logic is much harder to compute with than first-order logic. Indeed, the logic supported by ABASE is function-free Horn clauses for the most part, which particularly lends itself to efficient implementation. In the representation language, therefore, we do not treat "weight" as a function, but as a *constant* of type "property". Nor do we treat "head" as a function, but as a *constant* of type "component". Although properties and components are really functions and relations, treating them as individual objects of thought has many advantages. For example, a client of ASA could request it to list any properties which it can deduce a sample might possess but which have not yet been mentioned as being measured in this experiment.

Treating functions as constants is a standard trick for getting some of the expressiveness of second-order logic without leaving first-order logic. To make this work one has to introduce a new form which expresses the application of a function to its argument. In ASA there is an "application predicate" for properties, and one for components. For example, we would write

```
X is an object -> Y^(
    property(X, weight, Y) &
    Y is a physprop(mass,type_of(X))
).

X is a vertebrate -> Y^(
    component(X, head, Y) &
    Y is a head
).
```

The predication property(X,P,Y) represents the statement "Y is the P property of X". The predication component(X,C,Y) represents the statement "Y is a C component of X". Now there are some objects which are "components" of others in a sense, but are not physical components. For example, you brother, if you have one, acts like a component of you (is accessible through you, and is likely to share many properties with you), but is not physically contained in you. The predication part(X,C,Y) represents the statement "Y is a C component of X, and more than that, it is a physical part of X." 'part' is a specialisation of 'component', expressed by the rule

```
part(X, C, Y) -> component(X, C, Y).
```

For a given component constant, say 'limb', we can tell ABASE that all such components are parts by writing a rule like

```
X is an animal -> (
    component(X, limb, Y) -> part(X, limb, Y)
).
```

The fact that one object is a physical part of another object is important in Statistics. There are two reasons for this. One is that the mass (or volume) of a part is bounded above by the mass (volume) of the whole, so that the ratio of the two masses is a proportion. We tell ABASE about these ASA rules by writing

```
        physical_property(weight).
        physical_property(volume).

        part(X, _, Y) &
        physical_property(P) &
        property(X, P, WX) &
        property(Y, P, WY)
->      bounded_above_by(WX, WY).

        X is a physprop &
        bounded_above_by(X, Y) &
        measures(MX, X) &
        measures(MY, Y) &
        quotient(MX, MY, Pr)            % Pr = MX/MY
->      Pr is a proportion.
```

Logical containment is important for counts for the same reason. The second reason is that parts of an object are typically subject to the same environmental influences as the whole object. If, for example, we are examining the effectiveness of three treatments for baldness, we would ideally like to apply each of the treatments to a different area of the same scalp. ABASE could have provided a special notation for those components which are proper physical parts, but since the description system in ABASE is based on lattices, and parts form a lattice, it was possible to represent containment using existing means.

2.3. Introducing Value Spaces

The bulk of this chapter is concerned with Value Spaces, that is, with the *range* of properties. Thus length is an individual property, and so is height; both map an object into a distance measure value space. Objects are arranged according to an object taxonomy. Value spaces may be arranged into a taxonomy of their own. But I have found no need to classify attributes per se in this way.

ASA's basic view of the world is that *objects* have *attributes*, and that statistical *variables* are *measurements* of these attributes. The distinction between an attribute and a measurement of it is not a purely philosophical one, important though the distinction is philosophically. It has an important practical aspect as well: an experiment (such as a calibration) may contain several variables measuring the same attribute of the same object.

In fact this view is astonishingly close to the Entity-Relation-Attribute [Chen 76, Chen 83]data model. Considering that the ERA model was developed to give a more natural semantic description of data than other data base models, perhaps it should not be so surprising. The coincidence of the names "entity" and "attribute" is just that, I only encountered the ERA model while writing this volume.

Inheritance is quite important in the entity/attribute/measurement model. Entity types are organised into a type system, and the knowledge engineer can specify attributes and their characteristics at whatever level of the type system is most appropriate, with more specific descriptions overriding less specific ones. (See Chapter 7.) For example, all and only physical objects have weight, and this can be stated just the once. We say this in ABASE by writing

```
X is a physical_object ->
    Y^(property(X, weight, Y) & ????).
```

where ???? stands for some statement about Y. Starting with section 2.5, this chapter explains the (symbolic representations of) *value spaces* which fill in this blank.

Value spaces correspond to types in programming languages, or more accurately to "dimensions" in physical equations. They describe two sorts of objects: attributes, and statistical variables. One of the most important rules in ASA is

```
measures(Attr, Var) ->
     (the value_space of Attr is VS <->
      the value_space of Var is VS).
```

whereby a statistical variable inherits a value space from the attribute it measures. When and if ASA represents measurement processes, this rule will have to be replaced by something which takes the approximate nature of measurement into account. And of course not all variables measure intrinsic attributes of entities.

An important thing to note is that (the symbolic representations of) value spaces in ASA are *not* objects in their own right. They can be be combined in various ways, and compared. They are elements of an infinite lattice. This in fact is the main reason for ABASE's seemingly complicated type system. It is intuitively obvious that the "type" of a variable is "like" the "type" of an object, even though the two do not overlap, and there are infinitely many variable types and typically only finitely many object types. The benefits of using a single mechanism for both are obvious. What was not obvious was that it could be done. Finding out how the types of statistical variables could be handled was vital to ASA.

The lattice of value spaces is computed from the lattice of object descriptions. To give a trivial example, count(<description>) is a value space for any *object*-description, and the lattice relation between two count spaces is the same as the relation between the two descriptions they are based on. Thus count(species=orange) and count(species=apple) are two *different* value spaces, neither of which subsumes the other. ASA *will* add apples to oranges, and the result is count(species=fruit), not because this is built into a table of value spaces, but because "fruit" is the common super type of apples and oranges in the object taxonomy.

The impossibility of expressing this sort of relation in MBASE held me back for quite some time.

2.4. The Traditional Classification of Value Spaces.

Statisticians commonly distinguish between six different "levels of measurement". The original scheme, presented in [Stevens 46] had four. Other classifications are sometimes used. The point of distinguishing between these different levels is to determine which statistical methods make sense; a statistical method should never be used if the calculations it performs (such as comparisons or additions or square roots or whatever) do not make sense on the data.

One of the major reasons for having a package front-end such as ASA is that present packages do not know what level of measurement their variables attain. To quote [SPSS 75]: "The computer does not know what level of measurement underlies the numbers it receives, and will process whatever numbers are fed into it. Thus, it is up to the user to determine whether a particular technique is suitable for his or her data." I do not mean to imply that SPSS is a bad package, on the contrary it is well thought of. But all too often the users of such package do not understand the jargon of Statistics well enough to classify their variables according to the traditional scheme. And I found to my surprise that the traditional classification is not *adequate* to discriminate between many methods.

The traditional scheme is based on the question "under what mathematical opera-

tions is this scale invariant". The scales which are commonly identified are nominal scales, partially ordered scales, ordinal scales, ordered-metric scales, interval scales, and ratio scales. We shall now examine each of these in turn.

2.4.1. Nominal scales.

The mathematical structure here is a finite set. That is, there are no order relations between the values and no operations on them. All we can do is to tell whether two values are the same or not, or to count the number of measurements with a particular value. Some examples are

- gender (male, female)
- political party (National, Labour, Social Credit, Values, NZ)
- which programming language a program was written in.

2.4.2. Partially ordered scales.

The mathematical structure here is a finite partial order. The direction of the order is of course arbitrary. My research for this chapter found that partial orders whose Hasse diagram is a tree are often mistaken for ordinal scales. Appendix B discusses these scales, and proposes the name "multi-metic scales" for them.

As an example of a partially ordered scale, consider Polynesian archaeology. There is a "centre" to Polynesian culture, with migration of people and ideas outward. The various Pacific islands can be regarded for some purposes as points in a partial order, where X<Y when there was communication from X to Y. As we traverse a chain in this partial order, we expect to see trends (e.g. pottery at all sites up to one, and on none thereafter). But the sites cannot be arranged into a linear order.

2.4.3. Ordinal scales.

The mathematical structure here is a total order. The SPSS manual cites social class as an ordinal scale (though I would regard it as a partial order). A common type of ordinal scale is a human judgement of the intensity of some phenomenon, such as the severity of a disease. Note that this is distinct from what the intensity *is*: the Beaufort Scale and the wind speed have quite different properties.

2.4.4. Ordered-metric scales.

These are usually explained as being scales where we can not only compare the values, but we can compare the *differences* in the values. This is not strictly accurate, because the differences as such are not explicitly represented. What we have is a total order on the values together with a total order on the *intervals*, that is, given any two intervals [A,B] and [C,D] we can tell whether one is "wider" than the other. Given any total order, inclusion defines a partial order on intervals. The novelty of an ordered metric scale is that this partial order can be refined into a total order.

Suppose we have an ordered metric scale with n values $v_1 ... v_n$. We can embed this scale into the interval [0,1] by solving the following set of inequalities:

$$v_1 = 0$$
$$v_{i+1} = v_i + d_i \text{ for } i = 1...n-1$$
$$d_i > 0 \text{ for } i = 1...n-1$$
$$v_n = 1$$

when $[v_i, v_j]$ is bigger than $[v_p, v_q]$:

$$d_i + ... + d_{j-1} > d_p + ... + d_{q-1}$$

The distinction between an ordered metric scale and an interval scale is that these inequalities may have more than one solution for an ordered metric scale.

Examination scores are sometimes cited as an example of ordered metric scales. There is a ranking A,B,C,D,E, and one sometimes encounters claims that say [A,B] is a narrower interval than [D,E]. Here of course there is usually an original mark in the range 0..100, and the letter score is an approximation to that mark. This seems to be characteristic of ordered metric scales: they are (or are approximations to) numeric scores which the experimenter cannot persuade himself are interval scales.

It is always valid to treat an ordered metric scale as if it were an ordinal scale.

2.4.5. Interval scales.

Interval scales are the weakest scales on which it makes sense to do arithmetic. That is, as well as a total ordering relation, we have a difference operation, and the differences can be added and compared. Interval measurements can be well represented by numbers, but a statistic computed on such a scale should be invariant under linear transformations (f(aX+b) = f(X)) or covariant (f(aX+b) = af(X) or af(X)+b).

In thermodynamics, internal energy is an interval measure, because the zero point is arbitrary. Temperature is more than an interval measure, because there is an absolute zero.

2.4.6. Ratio scales.

Ratio scales are interval scales with a natural zero, so that we can take ratios of values as well as differences. However, the size of the units (such as feet or metres) is still arbitrary, so statistics computed on these scales should be invariant under change of unit (f(aX) = f(X)) or covariant (f(aX) = af(X)).

2.4.7. Deficiencies of this scheme.

An obvious deficiency is its incompleteness: *counts* do not fit in anywhere. Counts are even stronger than ratio level, because the size of the unit is fixed. And yet it is not true that all the operations applicable to ratio scales are applicable to counts: the mean of the counts {1,5,2,3} is 2.75 which is not a count. Or rather, it would be fair to say that this classification does not distinguish between operations being *applicable* to a scale and a scale being *closed* under those operations. Absolute scales are sometimes added to the list; but counts and proportions are both absolute, yet means make sense for proportions but not for counts.

Another distinction which this classification fails to capture is the distinction between strictly positive scales such as the weights of physical objects and differences of such scales. This is a *very* important distinction. Anyone taking the logarithm of a statistical variable whose values might be zero or negative is asking for the trouble he will certainly get. Yet there is a text book on regression which does this very thing! I have no doubt the author checked scrupulously that the variable in question was a ratio variable, but that is not enough. A commonly assumed distribution is the Normal or Gaussian distribution, but a strictly positive variable cannot possibly have that distribution. Yet you'll see it used very often for just such variables. Indeed, Physics students being introduced to the topic of errors in measurements are all too often *taught* to assume that instrument errors follow the Normal distribution. (Assuming that the logarithms of the measurements follow the Normal distribution would make mathematical sense, would in many cases simplify the calculations, and would give much the same results.)

Another distinction which is not made is the distinction between linear and periodic measurements. For example, the current time measured as the number of seconds since 1/1/70 is a linear measurement, but the time of day as reported by a clock is periodic. If one takes the definitions above, clock time belongs to none of the scale types, as 1pm is both before and after 2pm. However, it is very easy to forget this, and to look at the clock time as seconds since midnight and think that of course this is ordered, whereupon one mistakes clock time for an interval scale, or even, should one forget that starting at midnight is arbitrary, for a ratio scale. The temptation to make this kind of mistake is very great if one is

presented with the above list of six types of scale and told that this is how statisticians classify variables. Of course statisticians *have* studied periodic scales, and several useful methods dealing with them are known.

An omission which is unfortunate for the human sciences is that permutations are not considered, probably because they are not normally represented by single numbers. Yet a common "measurement" in psychology is to ask the subject to rank a set in order of preference, and what you get out of this is precisely a permutation. There are not many methods for working with permutations, but there are *some*, in particular there are methods for combining permutations from several subjects to obtain an overall preference scale. The fact that we are dealing with permutations is often disguised by saying that we have a multivariate problem, where each subject reported the *rank* he assigned to each of the k objects, instead of saying that each subject reported a single permutation of the k objects. This fails to take into account the strong relationship between numbers in the same row (reported by the same subject) and the weak relation between numbers in the same column (pertaining to the same object). There's another problem: ranks do not fit into the classical scheme either. They are certainly ordered, but they're stronger than that. There is no transformation you can apply to a set of ranks that makes any sense other than leaving them alone or reversing them all. So ranks seem like counts. On the other hand, there is no arithmetic you can sensibly do with ranks either; while you can sensibly divide by a count you cannot sensibly divide by a rank.

Another missing distinction is between unbounded measures such as the distance between stars, "naturally" bounded measures such as the height of human beings (where there *is* a bound but we may not *know* it with any precision), and "formally" bounded measures such as proportions and probabilities.

The worst problem with the conventional scheme is that it is difficult for clients to apply it themselves. It is easy enough to tell whether a variable is *quantitative* (the numbers mean something *as* numbers) or *qualitative* (the numbers are just labels), or is it? What if the numbers are coarse approximations? If we have people's ages measured to the nearest 10 years, can we still consider that a ratio scale, or must we consider it as an ordinal scale? It does not help that the answer can be different for different methods.

The point of ASA's value spaces is to have a method of classification from which these distinctions can be *derived*, and which can be defined in terms that make sense to the client irrespective of whether he intends to perform statistical calculations on the results or not. The physicist's notion of "dimensions" is an obvious place to start.

2.5. Classes of Value Spaces.

The rest of this chapter describes some classes of value spaces. The basic idea is that the set of value spaces forms a lattice. Many of the lattice elements are incomparable (e.g. counts and classifications have nothing in common). This lattice has a very rich mathematical structure.

An important aspect of this lattice is that it is built using a small number of functions. In particular, there are several (partial) functions from the object taxonomy (itself a lattice) into the value space lattice, such as

```
count : object_type -> value_space
mass  : object_type -> value_space
age   : object_type -> value_space
```

These generalise physical dimensions.

There are also functions for combining value spaces.

Each value space is represented by a logical term. I define the lattice operations $\wedge$ and $\vee$ on these terms as each type of value space is introduced. The top of the lattice has changed around as different abstractions have proven more or less useful. The current scheme has four top-level classes:

- value_space: the top of the lattice of value spaces.
- discrete: all categorical measurements, including approximations. A subclass of this, ordered_discrete, has been present or absent depending on whether I included ordinal scales or not. Currently ordinal scales are not represented, and ordered_discrete is out.
- numeric: all value spaces on which arithmetic makes sense.
- positive: all strictly positive value spaces. Counts are a subclass of this.

Here is a diagram of the lattice of value spaces.

```
value_space
|
+==discrete
|   |
|   +==[partially_ordered]                    {Note 1}
|   |   |
|   |   +==place(Description)
|   |
|   +==[ordered_discrete]
|   |   |
|   |   +==[gorder(N)]
|   |       |
|   |       +==approx(VS, N)                  {Note 3}
|   |       |
|   |       +==[gorder(Description, N)]
|   |           |
|   |           +==[order(Description, V)]
|   |
|   +==[nominal]
|       |
|       +==gclass(Description, N)
|           |
|           +==gclass(Description, AttrType, V)
|               |
|               +==class(Description, Attribute, V)
|
+==numeric
    |
    +==positive
    |   |
    |   +==count(Description)                 {Note 4}
    |   |
    |   +==proportion(Desc1, Desc2)
    |   |
    |   +==physprop(Attribute, Description)
    |   |
    |   +==PosVS1*PosVS2                      {Note 5}
    |   |
    |   +==PosVS1/PosVS2
    |
```

```
+== -count(Description)
|   |
|   +==count(Description)
|
+== -proportion(Desc1, Desc2)
|   |
|   +==proportion(Desc1, Desc2)
|
+== -physprop(Attribute, Description)
|   |
|   +==physprop(Attribute, Description)
|
+== -(PosVS1*PosVS2)
|   |
|   +==PosVS1*PosVS2
|
+== -(PosVS1/PosVS2)
    |
    +==PosVS1/PosVS2
```

{Note 1} The [bracketed] lattice elements are just place-holders; ASA makes no actual use of them. In particular, approximations are currently treated as if they were unordered.

{Note 2} An **extremely** important fact about this lattice is that it is determined by the data. For each distinct subtype of "object", there is a corresponding distinct count(_) value space, and the count(_) sublattice of value spaces is isomorphic to the sublattice of object types (count(**0**) being defined as **0**).

{Note 3} The recursion here is only apparent. An approximation to an approximation is an approximation. Specifically,

```
approx(approx(VS,M),N) = approx(VS, min(M,N))
```

Note that this is not a rule expressed in ABASE, but is done by Prolog code which is "part of the machinery". The whole lattice of value spaces is managed by Prolog code. ABASE supplies some routines to manage lattice computations, but the fact that the knowledge engineer can supply his own computation rules is important.

{Note 4} count(_) appears twice, under "positive" and under -count(_). This is because it is a subtype of both types, neither of which is a subtype of the other. Each of the positive numeric value spaces is a subtype of positive and of -(itself). See the description of the - function.

{Note 5} The recursion here is real. The lattice of value spaces is therefore infinite.

Four points deserve stress:

- this is a *lattice*, not just a tree. It cannot be flattened into a tree without losing information. This was the biggest problem with the first prototype of ASA, which used MECHO's type system.

- the lattice is generated from the lattice of object descriptions, which is partly provided by the knowledge engineer and partly acquired at run-time.
- the lattice is infinite. The physicists' system of dimensional analysis is embedded in this lattice. Any "type system" for mathematical modelling needs to be able to cope with dimensions.
- it can be shown that any particular consultation will only deal with a finite set of lattice elements; calculations on lattice elements are not handled by general rules but by special code which means that the calculations are efficient and terminating.

2.6. Discrete Value Spaces

Recall that value spaces describe the range of properties in themselves, not our measurements of them. Age is a continuous property; if we choose to record an approximation of it with five levels that does not affect what an age *is*.

This may seem like quibbling, but in fact there are quite a number of statistical tests for discrete variables which assume that there is an underlying continuous property. In order to select appropriate methods for such variables we have to recognise that the property being measured is continuous, but to avoid selecting *in*appropriate methods we have to recognise that the variable is discrete. These methods work on the discrete variables *as* variables. It is not the nature of the calculations which demands an underlying continuous property. No, the question is whether the results can be interpreted.

2.6.1. Classifications

The most fundamental kind of measurement is to take a look at something and say what sort of thing you think it is. This is *classification*. Classification properties give rise to *nominal* variables.

The clearest case is when we have a simple type tree, and we are given some objects known to belong to a particular Taxon. If this Taxon has N immediate subtypes, Sub-1, ..., Sub-N, then classification will yield the name of one of these subtypes as the measurement. Some examples, taken from [Everitt 77], are

```
attribute gender (male, female).
        class(gender) is a binary value space.

attribute handedness (sinister, ambi, dexter).
        class(handedness) has three unordered values.
```

```
attribute voting (voter (national, labour, socred),
                  abstainer, unregistered).
        class(voting) has three unordered values.
```

The lattice of classifications

The basic description of a classification value space is

```
class(Description, Attribute, SetOfValues)
```

where Description is a consistent description[2], Attribute is the name of an attribute, and SetOfValues is a set of values belonging to that attribute, such that

```
for each V in SetOfValues,
        "the Attribute of X is V" is consistent with
        "X is Description".

for each V, W in SetOfValues such that V ≠ W,
        "the Attribute of X is V" is inconsistent with
        "the Attribute of X is W".
```

That is, each of the values must be a *possible* value of the attribute for objects already known to satisfy the description, and any object satisfying the description should be classified in at most one class. This generalises the definition given above, in that the Values (subtypes) need not be at the same level of the hierarchy. That is, if we have a full zoological taxonomy for vertebrates, we can accept

```
    class(species=vertebrate,
        species,
        [rabbit, horse, pig])
```

as specifying a classification, despite the fact that they are

```
rabbit: infra-class = eutheria, order = lagomorpha,
        genus = oryctolagus;

horse:  infra-class = eutheria, order = perissodactyla,
        sub-order = hippomorpha, genus = equus;

pig:    infra-class = eutheria, order = artiodactyla,
        sub-order = suiformes, genus = sus
```

respectively.

This raises a problem. The well-formedness condition requires the values to be consistent with the description. It does *not* require them to be exhaustive. It might so happen that in a particular experiment we have sampled three populations; one of rabbits, one of horses, and one of pigs, so that each subject in the combined sample *must* fall into one of these three classes. The representation of experiments used by ASA is such that in this particular case there is enough information to deduce that the classification is exhaustive for that experiment. Often, though, the classes listed will be exhaustive only because the experimenter means to discard any subject which does not belong to one of these classes, so

---

[2]see Chapter 4 for an explanation of what this means; for now think of it as a type in the usual sense

that the population he has in mind is just the three types. My representation language is incapable of representing this fact; if you say that the population contains rabbits, horses, and pigs, the whole *point* of the lattice-based type system is that it will generalise to infra-class=eutheria.

We have a lot of freedom in designing the lattice of classification value spaces. The lattice I present here seems to work, but I do not claim that it is more than convenient. There are three levels of description:

1. `gclass(D, N)`

is the weakest. It describes any classification of an object satisfying description D into one of N values. We have

```
gclass(D1, N1) \/ gclass(D2, N2) =
        gclass(D1\/D2, N1) if N1=N2
        'discrete' otherwise

gclass(D1, N1) /\ gclass(D2, N2) =
        gclass(D1/\D2, N1) if N1=N2
        0 otherwise
```

2. `aclass(D, DA, V)`

is the next weakest. It describes any classification of an object satisfying description D, where the attribute being inspected is not named, but satisfies description DA, and the range of possible values is V. This is useful for considering several similar classifications of the same individual. We have

```
aclass(D1, A1, V1) \/ aclass(D2, A2, V2) =
        aclass(D1\/D2, A1\/A2, V1) if V1=V2
        gclass(D1,#V1) \/ gclass(D2,#V2) otherwise

aclass(D1, A1, V1) /\ aclass(D2, A2, V2) =
        aclass(D1/\D2, A1/\A2, V1) if V1=V2
        2b<0> otherwise
```

where #V is the number of values in the set V.

3. `class(D, A, V)`

is the strongest, and was described near the beginning of this subsection. A is an attribute name. We have

```
class(D1, A1, V1) \/ class(D2, A2, V2) =
        class(D1\/D2, A1, V1) if A1=A2&V1=V2
        aclass(D1, @A1, V1) \/ aclass(D2, @A2, V2)
                                otherwise

class(D1, A1, V1) /\ class(D2, A2, V2) =
        class(D1/\D2, A1, V1) if A1=A2&V1=V2
        0 otherwise
```

where @A is the description that attribute A satisfies. (Since A is a constant at the time the knowledge base is built, @A can be determined then, rather than during a consultation.)

2.6.2. Approximations

For any continuous value space VS (even periodic ones like time of day), and for any integer N greater than 1, there is a value space

```
approx(VS, N)
```

If we were being really strict, we would want to know not only how many classes there were, but where the dividing lines were drawn. For example, the two measurements

- if more than 5 cigarettes smoked per day then 2 else 1
- if more than 20 cigarettes smoked per day then 2 else 1

are clearly different. Including the category limits in the description would lead to a lattice very similar to the lattice for classifications.

However, for the experiments I studied, the choice of category limits had no effect on which analysis method should be chosen. The number of categories *does* influence which analysis method is chosen: if there are more than say 20 categories treating the variable as if it were continuous is unlikely to lead to serious error, while if the number of categories is 2 there are several specialised techniques applicable.

```
approx(VS1, N1) \/ approx(VS2, N2) =
        approx(VS1\/VS2, N1) if N1=N2
        'discrete' otherwise

approx(VS1, N1) /\ approx(VS2, N2) =
        approx(VS1/\VS2, N1) if N1=N2
        0 otherwise

approx(approx(VS, N1), N2) =
        approx(VS, min(N1,N2))
```

These equations, like the other equations defining lattice operations in this chapter, are not presented by the knowledge engineer in the form of general rules, but as Prolog clauses. Here are the actual clauses:

```
lub(approx(VS1,N1), approx(VS2,N1), approx(VS,N1)) :- !,
        lub(VS1, VS2, VS).
lub(approx(_,_), approx(_,_), discrete).

glb(approx(VS1,N1), approx(VS2,N1), approx(VS,N1)) :- !,
        glb(VS1, VS2, VS).

normalise(approx(approx(VS,N1),N2), approx(VS,N)) :- !,
        min(N1, N2, N).
normalise(approx(VS,N), approx(VS,N)) :-
        ako(VS, numeric),
        !.
```

There are some global defaults: if glb/3 fails, the glb is taken to be **0**, the result of lub/3 or glb/3 is always normalised, if normalise/2 fails, the result is taken to be **0**. You are to understand all the rules for ∨ and ∧ and so on in this chapter as standing for Prolog clauses like this. Prolog seemed at least as clear as any special ad-hoc notation for this task.

Note that if approx(Dom, L1) and approx(Dom, L2) appear in the model, where L1 =/= L2 we are entitled to ask the client what he thinks he is playing at. An example of this happening in practice is a biological experiment that a friend of mine analysed. As blackbirds grow up, their eye-rings and beaks change colour. The yellowness of a blackbird's beak was measured on a five-point scale, while the yellowness of its eye-ring was measured on a 3-point scale. This meant that the two variables could not be compared! Simply taking a colour photograph of each bird, and comparing the relevant parts with a standard colour chart would have given a common scale for both.

We can express this check as a "negative rule" or demon:

```
<-  V1 is an approx(VS,N1) &
    V2 is an approx(VS,N2) &
    V1 is not an approx(VS,N2).
```

That is, if there are two statistical variables which are both approximations to some VS and they do not have exactly the same type (the third conjunct can only succeed if N1 and N2 are unequal), this demon will fire. In fact the rule should not be quite so strict, because the client may in fact have such data, so we would write

```
<-  V1 is an approx(VS,N1) &
    V2 is an approx(VS,N2) &
    V1 is not an approx(VS,N2) &
    {bad_approx_query(V1, V2)}.

bad_approx_query(V1, V2) :-
    describe(V1),
    describe(V2),
    write('You have two different approximations'), nl,
    write('to the same underlying value space.'), nl,
    \+ yesno('Did you really mean this').
```

Curly braces enclosing a conjunct mean that it is a Prolog goal. Such a goal is invoked once all its "input" arguments are known and type checked. Prolog escapes are very useful, but they are unfortunately invisible to the dependency maintenance subsystem.

I originally built this test into the normalise(approx(... clauses, but this was a bad idea, because it was not possible to correct the problem by modifying the description of the older variable, and it didn't allow the analysis of the blackbird experiment. One of the design goals of ASA from the start was to permit the specification of advice for the client like this.

When L1 = L2, we assume that approx(Dom,L1) and approx(Dom,L2) are the same. This is not necessarily true, but a problem where it is not is probably beyond mechanical aid.

2.6.3. Ordered Value Spaces

I have not been able to think of any ordered discrete value spaces (apart from counts, vide infra) which are not really approximations to some stronger value space. A day may be described as rainy, drizzly, or clear. But that is an approximation to "the amount of rainfall", and is not really a classification at all. A patient may be described as "well, poorly, sick, very ill, dangerously ill", but there is clearly something like "degree of risk/probability of unaided recovery" going on here which we would measure if we could. Some very popular human sciences properties are of this sort. Socio-economic status (class) is a portmanteau property which gives you a rough guess at the subject's schooling, nutrition, exposure to disease, ... all of which the researcher would rather have if only he could measure them. The ever-popular "number of years in secondary education" is another. A common feature of this sort of measurement is that it is a combination of several aspects of the subject, weighted no-one knows how, and has between 3 and 7 levels.

Here is an example from [Everitt 77]. The subjects are people, and the measurements are Age (a time) recorded as under/over 40 years, and the number of cigarettes smoked (count(cigarettes)/time) recorded as less/more than 20/day. These values can clearly be ordered, and if there were more classes it would not make much difference. In fact, it is almost certain that the age was recorded to the nearest 5 years or even more accurately, and the cigarettes/day recorded to the nearest 5/day or so, and the dichotomisation has been done for the benefit of the analysis. If we had the raw data, we might be able to fit a more specific model and make stronger conclusions.

An extreme case of this is where some feature of the subject is recorded as present or absent. Again from [Everitt 77], psychiatric patients were classified as psychotics or neurotics, and examined to see whether or not suicidal feelings were present. The first variable is clearly a classification, and the second variable a dichotomisation of "degree to which suicidal feelings are present". At first sight, it looks as though we have an illustration of the fact that class(X) variables are plausible explanatory variables. Unhappily, the presence or otherwise of suicidal feelings may well have been used to diagnose these patients, so it is as legitimate to treat it as explanatory and "diagnosis" as that to be explained.

The best way to treat so-called "ordinal" variables that I can think of is to explicitly introduce the value space being approximated, and to include in the model the fact that the property being investigated has values in that value space, then to introduce approx(Domain, Levels) as the value space of the outcome of the measurement process. Some properties should be inherited from Domain, but others from the fact that the approximation is discrete.

At times I have included ordinal scales directly in ASA. This has been done by having

```
gorder(Description, NumberOfValues)

order(Description, SetOfValues)
```

points in the lattice, where SetOfValues is just a list of words. The reason that 'order' is not in the current version of ASA is the difficulty of finding an interpretation for the SetOfValues. In 'class', the SetOfValues is a set of taxa, a concept already in ABASE. But with 'order', the SetOfValues has no other connection with ASA's knowledge.

In the experiments I have studied, omitting 'order' has only caused trouble with socio-economic status.

2.6.4. Locations

If we define a "place" as being a compact[3] subset of the three-dimensional universe, it is clear that places cannot in general be arranged in a linear order. They can however be ordered by inclusion; Hope Park Square is included in Edinburgh which is included in Scotland. So locations are an instance of *partially ordered scales*. In fact they are something stronger: taking union for $\vee$ and intersection for $\wedge$, it is a theorem of topology that places form a lattice. This is not to say that any particular list of named places is a lattice; some of the unions and intersections might not have names. So places in general are lattices, a set of named places is a partial order.

At first sight, experiments are seldom concerned with overlapping places. We might record which of three districts a particular rat was trapped in, or which of several schools a class came from. It might seem that locations were basically just a form of nominal variable. However, a great many experiments are vitally concerned with overlapping places. Agricultural experiments often have this character. We might have three experimental farms in different parts of the country. In each farm, we might have a dozen fields. In each field, we might have 50 plots, viewed as 10 columns by 5 rows. The inclusion relations between these places are important.

place(Description) is a value space provided "X is a place" is consistent with "X is Description".

place(D1) $\wedge$ place(D2) = place(D1 $\wedge$ D2)

place(D1) $\vee$ place(D2) = place(D1 $\vee$ D2)

We've now finished with discrete value spaces.

---

[3]closed and bounded

2.7. Numeric Value Spaces

2.7.1. Counts

Counts are very important indeed. Even in an experiment where counts are not explicitly mentioned, they turn up during the calculations. For example, if we calculate the sample probability of something, that is a ratio of counts.

For any description, count(Description) represents the value space of counts of entities satisfying that description.

count(D1) $\wedge$ count(D2) = count(D1 $\wedge$ D2)

count(D1) $\vee$ count(D2) = count(D1 $\vee$ D2)

Counts are quite strange in many respects. They are discrete yet numeric. They are positive, yet unlike most positive attributes a count of 0 makes sense and may actually be measured. There are methods (such as the double root transform or taking square roots) for transforming counts into general numeric measurements with pleasant properties (such as approximating the normal distribution).

Counts are fundamental in the analysis of discrete data.

2.7.2. Physical Properties

Physical measurements have many dimensions:

- mass
- length
- time
- temperature
- electric current (or charge)
- angle
- solid angle
- baryon number
- spin
- ...

Thus one measurement might be described as having dimensions

```
[(baryon number)/(solid angle)(time)],
```

and another as having dimensions

```
[(mass)(length)/(time)^2].
```

Many of these dimensions are fairly exotic, such as "spin". Electric current can be regarded as derived from electron flux, so can be described in my notation as

```
-count(unit_charge)/physprop(time,true)
```

The dimensions of main interest in Statistics are mass, length, time, and temperature. This is not a fundamental characteristic of Statistics as such, but a reflection of its origins and present use in the study of biological systems. Psychological experiments are much concerned with constructed scores such as measurements of "aggression", "orality", and the like, each of these scores constitutes another dimension.

An interesting thing is that physics (and conventional descriptions of statistical measurements) fail to distinguish between basic physical measurements and *differences* of physical measurements. In fact, the only reasonably direct measurements I can call to mind which yield negative as well as positive values are measurements of voltage, current, and the like. If we measure the mass of some object, we never ever obtain a negative number. But the difference between the masses of two objects may be either positive or negative. Similarly, if we are measuring radioactive decay, we might use a Geiger counter, and no matter how long or short a time we operate the counter it will never indicate a negative number of events. No human being has a height of -72", not even dead ones.

It is very important to distinguish between measurements which are strictly positive (such as heights, masses, ages) and measurements which can take either sign. It makes sense to take logarithms of the former, but not the latter. The latter may be modelled by a Normal distribution but not the former.

To allow for the fact that it may be convenient to take some other measurements as primitive dimensions, I have a class of entities called "physical_dimensions". In ASA, only 'mass', 'length', 'time', and 'temperature' belong to this class. For any object description D and physical_dimension P,

```
physprop(P, D)
```

is a value space containing only positive values. The order relations are

```
physprop(P1, D1) \/ physprop(P2, D2) =
        physprop(P1, D1\/D2) if P1=P2
        'positive' otherwise.

physprop(P1, D1) /\ physprop(P2, D2) =
        physprop(P1, D1/\D2) if P1=P2
        0 otherwise.
```

Like all the value space constructors, physprop is strict in its description argument. That is, physprop(P,**0**)=**0** for any P.

ASA does not really understand time. Time as a measurement is treated exactly like length as a measurement, and the only distinction currently made between the duration of some activity and the interval between two events is that one is strictly positive and the other is a difference. Time within an experiment is represented only as structure; ASA does not reason about the time it takes to perform an experiment. If we are measuring the heights of children, it makes a difference whether the children are in the experiment for five hours or five years. ASA is blind to this difference.

2.7.3. Directions

An extremely important class of measurements, which is missing from the classical set and is totally ignored by most packages, is *directions*. A direction is simply a point in $\mathbf{S}^n$, the n-dimensional sphere. In the plane, a direction is an ordinary angle. An experiment recording the direction in which various pigeons from the same cot fly would produce such a measurement. In space, a direction might be recorded as angle and elevation, but it is nevertheless a single entity, and analysing the two recorded numbers separately would be a mistake.

ASA does not know any methods for handling directions. But that is not the point. The point is that treating a direction (a cyclic measure) as if it were a linear measure like a distance is totally wrong. The idea of ASA is not to analyse every kind of experiment, but to analyse some experiments, never to produce an invalid analysis, and to have at least been useful as a preliminary screening when it cannot handle a problem.

In fact there *are* several methods known for handling cyclic data, and the only reason that no such methods were added to ASA's method base was that no new knowledge representation or inference principles seemed to be required.

Directions are continuous, and arithmetic operations are definable on them, but they are not ordered.

2.8. Derived Scales.

We can do arithmetic on numeric measurements. However, the results may have different properties from the original measurements. Indeed, we often perform transformations to exploit this.

2.8.1. Addition

A basic assumption that I have made is that the sum of two measurements of a given type belongs to that type. This is expressed as the rule

```
the value_space of Sum is VS <-
sum(A, B, Sum) &
the value_space of A is VS &
the value_space of B is VS.
```

This is not always strictly true. For example, if V1 is the length of a subject's left leg, and V2 is the length of the subject's right leg, then this would ascribe the type "length of (some) leg" to V1+V2. This of course is not right; we might have a rule that says that the length of human legs is bounded above by 1m (or whatever), but the sum of two such numbers could easily exceed this bound. The answer to this is that bound information should be inherited from properties to the variables that measure them, and not from one variable to another. In fact ASA does not handle bounds, so the problem did not arise. If it did handle bounds, we would have to interpret a type such as "length of leg" not as meaning the length of some particular leg, but as standing for any weighted combination of such measurements with positive weights. Indeed, we have to use such an interpretation if we are to regard the mean or standard deviation of a variable as having the same type as the variable.

2.8.2. Differences

If D1 is a positive value space, -(D1) is the value space of differences, obtained by taking two variables over D1 and subtracting one from the other. The interesting thing about -(D1) is that the 0 point is meaningful. So if D1 is a general numeric value space, -(D1) is also defined. Even if D1 has no meaningful 0 point, -(D1) has. The question is, whether there *are* any numeric value spaces without a meaningful 0 point. It is customary to cite temperatures as a counter-example, but they are not. The Celsius temperature scale does have a meaningful 0, it just happens to be -273.16 degrees C. So interpreted, temperatures are in fact strictly positive. Other seeming counter-examples are usually better regarded as ordered things that have been arbitrarily mapped to numbers. We must be careful to treat cyclic numeric value spaces (angles and dates) specially: they have no zero point, but neither is addition defined on them. (Differences are.)

The sum or difference of two differences is a sum *of* differences, so we do not need to iterate the difference constructor.

```
-(D1) /\ -(D2) = -(D1 /\ D2).
-(D1) \/ -(D2) = -(D1 \/ D2).
```

Differences are the first example where we see operations on value space representations other than the lattice operations. There is a unary function, -, which satisfies

```
- (D)   = -(D)
- (-D)  = -(D)
```

for D a positive value space. (The notation is similar to arithmetic, where -2 is a negative number, as -(D) represents a value space of differences, while - -2 is the unary minus applied to the negative number -2, as - (D) represents the operator - applied to the value space D.) The "differences of" operator '-' is obviously monotone in its argument.

We still have to work out what the lattice relations between a positive value space and its differences are. Fortunately, inclusion gives us the answer. If D is a positive value space, then

```
D /\ -D = D
D \/ -D = -D
```

That is, is something is both strictly positive *and* a difference, then it is strictly positive, while if it is strictly positive *or* a difference, it is at least a difference.

This lets us say what the type of a difference is in one rule:

```
    difference(A, B, Diff) &
    the value_space of A is V0 &
    the value_space of B is V0 &
    VS ≤ -V0
->  the value_space of Diff is VS
```

If we have the following variables:

```
the value_space of k1 is count(species=pig).
the value_space of k2 is -count(species=hippopotamus).
difference(k1, k2, k3).
```

it follows by the definitions of Λ and V given so far, together with the axioms of lattice theory, that

```
the value_space of k1 is -count(species=artiodactyl) &
the value_space of k2 is -count(species=artiodactyl)
```

is true, from which we can deduce that

```
the value_space of k3 is -count(species=artiodactyl).
```

We would obtain the same result even if k2 was not a difference.

The fact that we can write this rule so simply is pleasant. But it might look as though it does something strange to our type system. It forces us to regard -D as being vaguer than D, that is if we deduce that the value space of some variable is -D at one time, and D at a later time, that's ok and the answer is D. Yet when people write experiments up in textbooks, we generally find the physical dimensions of a variable mentioned first, as the more general information, and only learn that it is a difference when we see how it is calculated. So it would appear that being a difference is a more specific property.

However, the paradox is only apparent. For a difference could well be positive. Indeed, if we consider the expression V1+V2-V1 where V1 and V2 are both strictly positive, the expression must be strictly positive, yet it will be given a difference type. Less trivially, if V1 (say the body weight of an animal) is necessarily greater than V2 (say the brain weight of the animal), it is obvious that V1-V2 must be strictly positive, yet it requires greater reasoning power than ASA possesses to detect this.

There is an unfortunate consequence of this. If a variable follows the normal distribution, it must be capable of taking on both positive and negative values. So if a variable is known to be strictly positive, it cannot possibly follow the normal distribution, and it would be a good idea to apply some sort of transformation. But the representation presented here provides no way of saying that negative values are known to be possible. We simply cannot say in ABASE:

```
if the value space of V is -D and
the value space of V is not D then
the normal distribution is a possible distribution for V
```

Instead, we have to code this as a default rule

```
the value space of V is numeric -?->
candidate_distribution(V, normal).
```

(if the value space of V is numeric, it is worth assuming that V might follow a Normal distribution, but more information may contradict this assumption) and a negative rule

```
the value space of V is positive &
candidate_distribution(V, normal) ->
false.
```

(it can never be the case that V is strictly positive and might have a Normal distribution).

This would be a serious defect, were it not for the fact that the normal distribution has to be a default anyway. There are lots of other things which can make the normal distribution a bad assumption. Nor is the assumption always indefensible: if the bulk of the measurements are concentrated far enough away from 0, the normal distribution may actually be quite a good numerical approximation to the true distribution of a positive variable, and the probability it assigns to negative values may be negligible.

If this proved to be a serious limitation, we could simply extend the lattice, having below -(VS) a subtype --(VS) of variables which are known not to be strictly positive.

2.8.3. Products

The basic dimensions of physics, M(ass), L(ength), T(ime), and I(current) correspond in my analysis to difference value spaces, -physprop(mass,true), -physprop(length,true), -physprop(time,true), and -count(charge)/physprop(time,true). Physical dimensional analysis considers products and powers of dimensions. So must we.

If D1 and D2 are positive value spaces, so is D1*D2. The laws are

```
(D1 * D2) /\ (D3 * D4) = (D1/\D3) * (D3/\D4)
(D1 * D2) \/ (D3 * D4) = (D1\/D3) * (D3\/D4)
(-D1) * D2 = -(D1 * D2)
D1 * (-D2) = -(D1 * D2)
D1 * D2 = D2 * D1 if and only if D1 = D2.
```

The last rule looks very strange. Unfortunately, if we try to make the product of value spaces commutative, we do not obtain a lattice. For consider

```
count(species = horse) * count(species = pig) /\
count(sex = male) * count(sex = female).
```

Is this to be

```
    count(species = horse & sex = male)  *
    count(species = pig & sex = female)
or
    count(species = horse & sex = female)  *
    count(species = pig & sex = male) ?
```

When D1 and D2 are incompatible, as when we have length*time, then they could commute, but when the client can give us vague descriptions we may never have enough information to decide whether they can commute or not. This is not particularly pleasant.

We can use D^k, where k is a known non-negative integer, as an abbreviation in source code for D*...*D.

At first sight it looks as though there would be something to gain by representing powers differently from products. For if k is even, D^k must be positive whether D is or not, while D*D might represent the product of two variables, one of which happens to be positive and the other negative. In practice, there is not all that much to gain; the main reason for being interested in powers is that taking the Nth root is often a useful transformation, and we are only interested in doing that when the result must be strictly positive. (Nth root is a function from positive numbers to positive numbers; it is *not* a function from positive numbers to reals.) Furthermore, if we cued "taking the square root" off length^2 and not off length*length, we would seldom or never consider taking the square root of an area.

2.8.4. Ratios

If D1 is a numeric value space, and D2 is a positive numeric value space, D1/D2 is also a value space. The idea that miles/gallon is a different value space from feet/second is an old and useful one. The practice in physics is to take feet/second as length.time^{-1} and miles/gallon as length^{-2}. I take this approach a step further, and do not cancel the dimensions. So instead of regarding miles/gallon as length^{-2}, I would regard it as length/length^3,

and would not consider the floor area of a house (length^2) as belonging to the same value space as the number of gallons of aviation fuel consumed per mile flown by an aeroplane (length^3/length). Furthermore, because the different amount(_)s are distinguished, distance/volume(petrol) is a different value space from distance/volume(water).

This is particularly important when we consider **proportions**, which have the general pattern D1/D2 with D1 and D2 being specialisations of the *same* value space. Thus we do not regard count(sheep)/count(sheep) as being the same as count(goats)/count(goats), though both are specialisations of count(herbivores)/count(herbivores). And a proportion based on counts can be distinguished from a proportion based on continuous measures, such as weight/weight. This is important because the two kinds of proportion have differences which may be vital: if we have a sample of N sheep, and are concerned with the proportion of the sample suffering from foot-rot, the statistic can only take the values {0,1/N,...,(N-1)/N,1}, whereas if we have the lung weight/body weight for a sheep, it may take on any value within a range I am not biologist enough to quote. If N is big, the difference can be ignored, but there is another effect. A proportion based on counts may well be measured without error. If the counts are rounded to multiples of say 10, we might as well treat them as continuous (but approximated), but it is not uncommon for counts in the hundreds to be accurate. On the contrary, it is rare for a continuous variable to be measured with more than 1% accuracy, and we should make it a general rule to ask what the accuracy is.

The calculation rules for ratios are similar to the rules for products. There is an interesting point. We would like to warn the client whenever there is the possibility of division by zero. But we cannot do this by looking at the value space: it is not possible to distinguish between (-VS1)/VS2 and VS1/(-VS2). Instead, the demon involved notices whenever a division is called for rather than when a ratio exists:

```
<-  quotient(X, Y, Z) &
    Y is not positive &
    {warn_about_zero_divide(X, Y, Z)}.

warn_about_zero_divide(X, Y, Z) :-
    describe(X),
    describe(Y),
    describe(Z),
    write(Z = X/Y), nl,
    write('Watch out for division by zero.'), nl,
    fail.
```

There is a hack here: the Prolog escape is just used to print a warning message, and never succeeds.

2.8.5. Proportions.

The proportion of subjects who are in favour of cruise missiles is obviously a ratio:

```
count(human&wants_cruise=yes)/count(human).
```

So is the concentration of hydrochloric acid in a solution:

```
mass(fluid&substance=HCl)/mass(fluid).
```

But knowing that these values belong to these ratio types only tells us that they are positive. Proportions are bounded above by 1. They are very important in Statistics, as probabilities are proportions. There are transformations which one would only apply to proportions.

The difference between a proportion and a ratio is that the numerator of a proportion is contained in the denominator; whatever the numerator measures is part of what the denominator measures. There are many cases where the form of the experiment makes it easy to detect this.

As the second example shows, whether something is a proportion or not is relevant to attributes, unlike the distinction between continuous attributes and measurements which approximate them discretely.

So for any two "amount" value spaces D1, D2 where D2 includes D1, proportion(D1,D2) represents a value space. The rules for combining proportions using the lattice operations have to maintain this property of the numerator D1 being a special case of the denominator D2, hence all the extra /\s in the first argument of proportion. The rules are

```
proportion(D1,D2) \/ proportion(D3,D4) =
        proportion((D1\/D3)/\(D2\/D4), D2\/D4)

proportion(D1,D2) \/ D3/D4 =
        (D1\/D3)/(D2\/D4)

proportion(D1,D2) /\ proportion(D3,D4) =
        proportion(D1/\D3, D2/\D4)

proportion(D1,D2) /\ D3/D4 =
        proportion(D1/\D3/\D4, D2/\D4).
```

Similarly, when we calculate

```
D := proportion(D1,D2)
```

D gets bound to proportion(D1/\D2, D2).

It is possible to handle solid angles by regarding them as proportions. For example, instead of saying that a measurement was 1.5pi steradians, one can regard it as 3/8 of a sphere. This dodge works in any number of dimensions. In particular, there are two senses of "angle" which need to be distinguished. An angle which measures a *direction* is a cyclic

measurement; adding any multiple of 2pi to it leaves its meaning unchanged. But the angle subtended by some object is bounded below (by 0) and above (by 2pi) and so is best regarded as a proportion. As an example of the non-triviality of the difference: the directions 0 and 2pi are identical, but an empty sector and a full disc are dramatically different.

2.9. Pattern Matching and Lattices.

The simplest way to state a rule about addition is to say that the sum of two variables belongs to the same value space that they do. So one might write

```
sum(A, B, Sum) &
the value_space of A is VS &
the value_space of B is VS
->
the value_space of Sum is VS.
```

Given the following facts:

```
sum(v21, v37, v49).
the value_space of v21 is count(species=apple).
the value_space of v37 is count(species=orange).
```

we would expect to be able to deduce that

```
the value_space of v49 is count(species=fruit).
```

However, implementing this in the obvious fashion is not going to work. What we have to do is to take each repeated description, generate new variables for all the occurrences on the left hand side, and add a least-upper-bound function. Thus the rule as coded would be

```
sum(A, B, Sum) &
the value_space of A is VSa &
the value_space of B is VSb &
VS = VSa\/VSb
->
the value_space of Sum is VS.
```

Why did this problem fail to appear in MECHO? For the simple fact that MECHO rules could not contain variables whose values were descriptions, or at any rate were not supposed to.

2.10. Examples

This section presents two examples. They show how knowing exactly what sort of measurement you have can be useful.

ASA is meant to provide assistance to naive clients. While it lacks the linguistic and instructional skills necessary to cope with real clients, my main concern is with knowledge representation and use, so that is not currently a problem. One of the tasks consultants have

to perform is to assist the client to formulate his goals. Very often the client has no clear idea of what he expects to learn from his experiment. That is not always stupidity.

The current design of ASA provides a limited amount of assistance merely by having a taxonomy of goals, and letting the client walk through the taxonomy to find something he thinks useful. I have stated elsewhere that ASA is intended to eliminate the practice of blindly searching through the SPSS manual looking for a Procrustean bed into which the data may be forced. This appears to be re-introducing that very practice. In a sense it is, but the space of goals from which the client may choose is very small, and the nature of the goals should be intelligible to the client. When the client picks a method from the SPSS manual, he may be totally unaware of what goal the method is intended for.

One of the high-level goals which ASA recognises is fitting a model. This means expressing one of the client's variables in terms of other measured variables and an uncontrolled error. Naive clients are particularly likely to need help formulating such a model. A model must have two properties: there must be a statistical method which can be used to fit the model, and the model should make sense. The client is unlikely to know what models can be fitted, and it is amazing how many Statistics textbooks fit models which make no sense.

Of the two examples, one involves "modelling" (summarising) a single variable, and the other involves explaining one variable in terms of another.

2.10.1. Rain-drops Keep Falling on my Head

This example comes from page 51 of [Velleman 81], where the authors are explicitly describing the use of transformations to reshape a data set. The data are the amount of rainfall at Minneapolis/StPaul during March for 30 successive years. The goal is to summarise the data, and it happens that data are best summarised when their histogram is symmetric.

The task, then, is to find a "suitable" transformation for the data. Now the data are *volumes*, that is, the value space of the measurements is

```
length(water) * length(water) * length(water)
```

There are three rules relevant to this task. Simplified, they are

```
candidate_power(X, 1) <-

candidate_power(X, 0) <-
        the value_space of X is positive
```

```
candidate_power(X, P) <-
        the value_space of X is D &
        is_power(D, K) &
        reciprocal(K, P)
```

(This is not the place to discuss power transforms, or what the numbers mean. Suffice it that they are a standard family of transforms widely used in Statistics.)

The first rule says that the raw data are always interesting.

The second rule say that taking the logarithm (that is what power transform 0 does) is interesting when the data are strictly positive. Here "positive" does not stand for a particular measurement class; "length(water) is positive" is true even though the left-hand side is not an instance of the right-hand side. Chapter 4 explains how this works. The possibility of having generalisations like this which are not just syntactic generalisations is one of the major reasons for my new type system.

The third rule says that if the physical dimensions of some measurement are expressible as a Kth power of a simpler kind of measurement, taking the Kth root is interesting. The "is_power" calculation has to be programmed by hand; the possibility of integrating such calculations into the type system without destroying its monotonicity is also one of the great benefits of my new type system.

These three rules suggest summarising the raw data, the logarithm, and the cube root. It is important to realise that these rules were *not* created for this example, but are general rules applying to *all* transformation goals, and were first added to ASA to handle other examples. There are several other transformation methods known to ASA which are specialised for measurement types such as counts and proportions. They do not apply in this particular case.

It turns out, when applied to the data, that the cube root transform does very well indeed, the cube roots of the rainfalls being very close to a Normal(1.14,0.23) distribution.

It is interesting to quote from [Velleman 81]. We reached these three transformations by looking at the *type* of the measurement, not by looking at the numerical values. In [Velleman 81] the authors are describing how to pick a transformation by performing certain calculations on the numerical values. Presented with this problem, they "fire a transformational shot-gun" at it, trying about 6 different commonly useful transformations in the hope of finding a good one. They say

> We might try a power between [square-]root and log, such as the 1/4 power, but this batch has only 30 data values -- too few for such fine discriminations. If we had to choose among re-expressions listed in Exhibit 2-4 [the L-V displays], we might select the square root for its simplicity. (Some meteorologists have found the 1/3 power quite desirable.)

The point is, we got to the 1/3 power straight away, without being seduced by square roots or

retarded by reciprocals. We would certainly have looked at the log transformation, but might not have bothered fitting the raw data. So with 2 expressions to check, we would have found a transformation that meteorologists apparently like, while [Velleman 81] examined 4 expressions without finding it.

2.10.2. Welsberg's Brains

This example comes from [Weisberg 80], a well known text-book. The subjects are animals (in fact "typical" individuals from some 60 mammalian species), and the two measurements are the weights of these animals' brains and the gross weights of their bodies.

The two measurements, Brain and Body, both have type

```
physprop(mass, a brain of a mammal)
```

Since both variables are continuous, the kind of model we are looking for has the form

```
f(Brain) = a*g(Body) + b + Error
```

where f and g are some transformations, to be decided on by us, and a and b are numerical coefficients to be estimated by regression from the numerical results of the "experiment".

The same rules that were presented in the previous section apply again, though this time neither measurement is a power of anything. (If brain *volume* had been measured, that would have been a different story!) Accordingly, there are four models suggested:

```
(1) Brain = a*Body+b
(2) Brain = a*log(Body)+b
(3) Brain = b^(a*body)
(4) Brain = b*body^a
```

Only two of these models (1 and 4) are at all attractive. We can eliminate (2) and (3) by insisting that when two variables of similar type appear in a model they should undergo the same transformation. Variables of similar type will always have the same candidate transformations , so there is no difficulty in finding similar transformations.

We are thus led to models (1) {always try the raw data} and (4) quite directly. It happens in this example that model (4) is exactly the right model to look for. Indeed, there is a rule of thumb in biology [Peters 83] that in animals of the same general type and developmental stage, the size of two body parts will usually be related by a power law

```
<size of part 1> = b*<size of part 2>^b
```

There was no need to add this specific rule to ASA since it falls out of the general principle that taking the logarithm of strictly positive measurements is interesting.

2.10.3. Comment

Note that the information we needed in these two examples was the kind of information the client is likely to be able to supply. Instead of asking "is Rainfall a discrete/ratio/... variable", the client can say (clumsily) that it is a volume of water. All the statistical information we need about the variables can be deduced from this client-oriented specification.

2.11. Back to the Classics

There are three important facts about my classification of value spaces.

1. It is *client-oriented.* That is, the classification is not phrased in terms of statistical words or concepts, but is a simple extension of everyday dimensional analysis. The kinds of measurement are sufficiently distinct (which one out of a classification, approximations, counts, direct physical measurements, differences, products, ratios, proportions) that a client can easily identify the appropriate type.
2. It is *lattice-based.* This means that the methods of chapter 4 for reasoning with descriptions can be applied to value spaces. In particular, we can record vague information about a value space initially (such as the fact that it is numeric) and only ask for more detailed information when it is useful to know. This helps the client further, as he does not have to supply a complete description of a value space in one step.
3. It is *adequate.* That is, the usual classification can be recovered from it, as can distinctions (such as a discrete variable being based on an underlying continuous scale) which the usual classification does not make.

The third point is easy to demonstrate.

2.11.1. Nominal Scales

These are class(_,_,_) and its generalisations.

2.11.2. Partial Orders

ASA currently knows no methods for such variables. Textbooks do not commonly describe such methods, and while it is possible to adapt other methods (by techniques such as considering each chain of the order separately) the reasoning required is beyond ASA's capabilities.

This establishes that my lattice of value spaces is *incomplete*. There would be no difficulty in extending the lattice to include something corresponding to partial orders, if only there were methods that could benefit from it.

Nor does ASA have a representation for multi-metic scales.

2.11.3. Ordinal Scales

These are approx(_,_). In versions of ASA with order(_,_) in the lattice, they correspond to ordinal scales as well.

2.11.4. Ordered Metric Scales

ASA currently knows no methods for such variables. As I have yet to encounter an experiment with a credible ordered metric variable, this has been no handicap.

2.11.5. Interval Scales

Interval scales are numeric. None of the numeric value spaces understood by ASA corresponds to an interval scale; all of them are at least ratio. In a positive value space, the zero represents a real lower bound. In a difference, zero represents "no change", which is again a real possibility.

If you have an interval scale, the differences of pairs of values from that scale constitute a ratio scale. So an interval scale is pretty close to a ratio scale anyway. What kinds of measurements manage to be interval scales yet fail to be ratio scales?

One example is internal energy in thermodynamics. But you cannot measure internal energy, only changes in it.

A popular example is temperature measured on the Celsius scale. But temperature is a physical property, hence is strictly positive. Where have I gone wrong?

The answer is: nowhere. A variable which records temperature really does measure a physical property, and really does have a natural zero point in its scale. It is therefore meaningful to take quotients of temperatures. *But that doesn't mean you can do it with the raw numbers!* If you want to take the ratio of variables T1,T2 where both record temperature in the Celsius scale, you have to calculate

```
(T1+273.16)/(T2+273.16)
```

The lesson here is that there is a distinction between the properties enjoyed by a

variable because of what it *is* and the properties enjoyed by the *numbers* because of the way they have been *encoded*.

ASA never deals with the actual data, so it fails to make this distinction. It is only concerned with the abstract properties of measurements and variables, not with the properties of encoded numbers. A full-scale "intelligent front end" would need to make this distinction, and others.

So ASA has no value spaces which correspond to interval scales, because such scales either apply to unobservable properties or are an artefact of encoding, which can be avoided.

2.11.6. Ratio Scales

All numeric value spaces except counts and proportions correspond to ratio scales.

2.11.7. Absolute Scales

Counts and proportions are absolute scales.

2.11.8. Summary

Each definite value space in ASA can be assigned to exactly one of the conventional levels of measurement. There are scale types (partial orders, ordered metric, interval) to which no ASA value space is assigned. If and when ASA is extended with methods specific to those scales, the lattice can be extended.

The point for the client is that if ASA needs to know whether a variable is measured on at least an ordinal scale, it can instead ask for a description in client-oriented terms, and can deduce that an approximation or a numeric value is ordered.

2.12. Summary

This chapter presented and criticised the conventional classification of statistical variables. In its place, I offered an approach extending the idea of physical dimensions. The algebra which results is also a lattice: some measurement types are instances of more general measurement types.

Two examples showed the relevance of this information.

My search for a way to handle this knowledge in ASA led directly to the type system described in chapter 4.

Chapter 3
The Structure of Statistical Experiments

3.1. Outline

In this chapter I argue that an Expert System for Statistics *needs* a way of representing and reasoning about the *structure* of experiments, as well as the varieties of measurements. I give examples to show that information which can easily be deduced from the structure of an experiment is vital to the synthesis of a valid analysis.

I present a specific notation for describing the structure of experiments. This notation is incomplete. There are many kinds of experiments it cannot handle, and many subtleties that it fails to depict. This would be a serious deficiency but for two things. The first is that the notation as presented is adequate for the limited simple experiments which I intended to cover. Note that even statistical textbooks sometimes make mistakes with just such simple experiments.

The second reason why the incompleteness of the notation is not a serious deficiency is that the notation presented is merely an *example*. The point of the chapter is to present and illustrate an *approach* to the development of such a notation. The notion that an experiment can be described *as if* it were a mathematical function on sequences is the key to the whole chapter. Each particular element of the notation was developed to handle a class of experiments or methods. The notation had to be expanded, the foundation needed no such revision.

This chapter contains many examples of the notation. The details of punctuation are borrowed from mathematical computer science, but are not important. ASA does not use this notation directly; appendix A explains the way that the structure of an experiment is stored in ASA. The really important thing in this chapter is the idea that an experiment can be given a compositional "semantics" in much the same way as a computer program (see [Gordon 79]).

Currently, formal descriptive notations for limited classes of experiments are known, particularly for so-called "factorial" designs. (See [GLIM 78].) But there is no notation available for even such simple experiments as one finds described in [Velleman 81]. The very idea of such a notation, as well as the approach and the specific examples in this chapter,

appears to be original. I can testify that the discipline of writing down a formal experiment description is an excellent method for ensuring that ambiguities and omissions do not go un-noticed.

The notation permits us to start with a vague description of an experiment and refine it as we need to. Accordingly, the notation should be good for *designing* experiments as well as analysing them.

It is useful and traditional to distinguish two kinds of variables, treatments and measurements, depending on whether the experimenter controlled the value of the variable or not. I show in section 3.12 that there is yet a third kind of variable, identifiers. This was not something I put into the notation to start with, but a discovery which was forced on me by trying to be clear about the relation between experiments and data sets.

The contribution of this chapter to Artificial Intelligence is that it presents domain knowledge which should be represented. It serves as motivation for the knowledge representation techniques presented in the second half of this volume.

Here is a summary of the notation introduced for describing experiments. What we have is both a recursively defined lattice of *steps* and a structure of *functions*. Each generic lattice element is preceded by the number of the section in which it is defined.

```
3.6.  vague[Vo;Vt]
      |
3.7.  +==observe[Vo]
      |  |
3.5.  |  +==identity
      |
3.8.  +==then[S1;S2]
      |
3.9   +==separate[C1->S1, ..., Cn->Sn]
      |
3.10  +==assign[C; V; K1->S1, ..., Kn->Sn]
      |
3.11. +==select[V; K1->S1|P1, ..., Kn->Sn|Pn]
      |
3.12  +==disperse[C; S]
      |
3.13. +==when[V; T1->R1, ..., Tm->Rm; K1->S1, ..., Kn->Sn]
         |
         +==include[T; S]
```

In this table,

```
Cx is a component name (often the name of a set)
Sx is a step, which is why the lattice is recursive
Vx is a variable or a set of variables
Kx is a constant, a possible value of its associated V
Px is a probability
Tx is a test (boolean function)
Rx is a set of constants
```

Arguments are separated by commas when their order doesn't matter, by semicolons when the order does matter. In one view, names like 'observe' or 'disperse' denote functions from component names, variable names, constants, and steps into the lattice of steps. In fact, they are the generators of the lattice. In another view, these names denote higher-order functions, which is why their arguments are enclosed in square brackets. If you think of a step as a node in a graph, with arcs labelled by constants Ki or component names Ci to its substeps Si, and the node itself labelled by its type and associated variables, you won't go far wrong. The point about this being a lattice is that we can take a vague[] step and refine it without disturbing any of our existing conclusions.

It is important for you to realise that the particular lattice that ASA uses isn't all that special. Most of these step types should be included in any plausible algebra of experiments. However, the catalogue was developed as need dictated. Far from the catalogue being closed, it is of the utmost importance that this approach facilitates the graceful expansion of the catalogue. In the present system, each step is a function which takes a single stream of units to a stream of "investigated" units. This limitation is not fundamental: there are experiment structures where it is useful to consider steps which combine two or more streams. ASA's data structures can handle that. What is really vital here is the idea is that an experiment is made up of streams of "units" with "steps" which map a definite number of input streams to a definite number of output streams, that the structure of functions can be regarded as a point in a lattice, and that this structure can be incrementally refined in an inference-driven way. The metaphor of the Little Man with his medical history cards (section 3.3) is almost the entire point of this chapter, "the rest is commentary".

3.2. Why do we care?

The first thing to discuss is why we need a formal representation of the structure of experiments at all. The previous chapter showed that just knowing what sorts of things the variables measure is often enough for an analysis. What more is there one might need to know?

The first thing that comes to mind is the distinction between variables which measure the type or level of some *treatment* (that is, they are under the experimenter's control; he gets to decide what the values will be) and those which are the result of some *observation* (the experimenter has no control over the values). Consider the following three experiments:

A. We distribute a number of identical cars among a dozen drivers, and at the end of a week find out M, the number of miles travelled during the week, and G, the number of gallons of petrol consumed during the week.

B. We take the same cars and drivers, fill the tanks, and get the drivers to drive the cars

over a course of known length. For reasons that are not important here, it is helpful if they travel different distances, so we ask some of the drivers to drive once over the course, some twice, some thrice, and record the distance they travel in miles as M. Afterwards we measure G, the number of gallons of petrol consumed.

C. We take the same cars and drivers, and fill the tanks to possibly differing levels G. We then get the drivers to drive around the course until they run out of petrol, and record the distance travelled.

In all three cases, the variables are of exactly the same type; G is an amount of liquid (by volume), and M is a distance. Yet the models we should fit are different in each case:

```
A:  (G-errorG) = a + b*(M-errorM)
B:  (G-errorG) = a + b*M
C:  G = a + b*(M-errorM)
```

This example shows that knowing the types of the measurements is not enough. But in this particular case, it is enough to know which of the variables are determined by the experimenter, and which by nature, so perhaps we could manage with just one more fact about each variable. But there is still more. Consider the following experiments, where the point of the experiments is to determine the effect of watering (different amounts of water being added to equally sized pots equally dry initially) on the amount of water lost by certain plants during a one hour interval.

D. The experimenter first applies the 0ml treatments, then the 1ml treatments, then the 2ml treatments, and so on.

The obvious snag here is that the amount of water added is confounded with the time of day. That is, since the time of day determines the amount of water added, it is impossible to tell whether a plant was affected by the amount of water, or by something else controlled by the time of day, such as the amount of sunlight. A statistical consultant should ask whether treatments were applied according to a fixed plan in order to warn about the danger of such confounding.

E. The experimenter applied a 0ml treatment, then a 1ml, ... then a 10ml, and kept repeating this cycle until there were no pots left to treat.

This is a more subtle form of D.

F. The experimenter applied the treatments in any old order that came into his head. ("haphazard" order)

This is an all too common practice. The experimenter is likely to be conscientious in avoid direct connection between the treatment and any obvious environmental vari-

ables, so that the obvious errors of D and E are avoided. But all this means is that there could be other factors operating, and the haphazard assignment ensures that we shall not know what those factors might be. Maybe we are safe, but we cannot be sure.

G. The order of the treatments was randomised.

This is very helpful, because we can apply statistical tests of significance (called "randomisation tests") even if the plants themselves do not form a random sample from any population.

H. The time of day and the amount of water were considered (and recorded!) as two factors, and the treatment plan was set up as a Latin square or some such.

This is even better. We can evaluate the time of day effect as well as the watering effect.

The analysis proceeds quite differently in cases H and G and the rest. So it is not enough to know that the treatments were under the experimenter's control, we must also know how that control was exercised.

But still, this amounts to more knowledge about single variables. Why should we care about the structural ways in which the variables are related?

The reason is that there are relations *between* variables which are important. One **extremely** important relation is whether two variables were measured on the same subjects or not. Another is causal flow. Yet another is statistical independence. We could indeed ask the client about each interesting combination of variables, but very often the questions will be unintelligible to him or her. **The structure of the experiment, referring as it does to the brute question "and then what did you *do*", does lie within the client's competence**, so deducing inter-variable relations from the structure is worth doing.

This is a key point! The client is in all probability ignorant of Statistics. He is most unlikely to understand clearly what the relations mean; confusing statistical independence with lack of (known) causal connection is a very common mistake. Simply asking the client about each of the relations between the variables is a very good way to confuse him and to obtain incorrect answers. But the client *does* know what he did. The reason for developing a notation for the structure of experiments regarded as planned procedural activities is simply that we can obtain reliable information about them using that view.

First let's consider statistical independence. Clearly, if the experimenter has chosen a treatment level at random, it will be independent of all the other measurements made on that object *up to that time*. However, it will *not* be independent of subsequent measurements,

in the normal sense of the word, and its statistical independence of them is something we might be interested in checking but in general *must* not assume. Consider the following little experiment:

I. The experimenter has a sample of apples. He measures the sugar content of each apple by testing some of the flesh. He then divides the apples at random into five equally sized groups, storing them at five different temperatures for four days. He measures the sugar content of each apple again.

Here we have three variables:

```
sugar0 : measured amount of sugar by weight
         (before treatment)
temp   : randomly assigned temperature
sugar1 : measured amount of sugar by weight
         (after treatment)
```

Clearly temp and sugar0 are necessarily independent, though in any one particular instance of the experiment they might be correlated in that sample, but it is most unlikely that sugar1 and sugar0 are independent, or that sugar1 and temp are. In fact we might fit a model

```
sugar1 = f(temp)*sugar0 + error
where (∀X) 0 ≤ f(X) ≤ 1
```

But if we only look at what the variables measure, and whether they are observations or treatments, we will have no way of telling that this model is a reasonable one while

```
sugar0 = g(temp)*sugar1 + error'
```

is less reasonable.

It might be argued that this is the client's job; it is up to him to say which variable he wants explained and which variables he wants to use in the explanation. Clearly a program which prevents a client performing an analysis which he deliberately chooses to do is a bad program, but a program which fails to comment on the strangeness of trying to explain an observation in terms of the experimenter's *later* decisions is also a bad program.

From the mere fact that these variables are observations and those are treatments, it follows that it is never sensible to look for an "explanation" (model) of those variables[4]. But that is not enough to tell us *which* of the treatment variables it makes sense to include in a model for one of the measurements. For that we need to know the order in which the treatments were applied, as it makes sense to include only those treatments which were applied *before* the measurement was made. We can and usually should include other

---

[4]this does *not* mean that it is never useful to perform the calculations that are normally associated with constructing such a model. It is often a good idea to double check that the treatments *were* in fact independent of the data, as this can uncover recording errors (e.g. the coding form and the program disagreeing about which variable goes in what column) or faults of procedure. But while the form of the calculations may be the same, the *purpose* and *interpretation* are different. A simple-minded program such as ASA does not bother with such checks.

measurements in a model as well. If we measure A, apply a treatment, and then measure B, it makes sense to include A in a model for B. Including B in a model for A is much less sensible, though sometimes it is necessary.

Continuing example I, then, these two rules [Rule 1] never look for an explanation of a treatment

[Rule 2] when forming a model for a measurement, exclude subsequent treatments and measurements following subsequent treatments produce the following behaviour:

(a) there are no other variables we could include in a model for sugar0. Therefore a model for sugar0 is a distribution, which we might report as a standard distribution name together with its parameters, or as a cumulative frequency table, or as a graphical display. sugar0 is necessarily positive, so displaying

(a1) the raw data (stem-and-leaf)

(a2) the logs of the raw data with a smoothed curve

(a3) a quantile-quantile plot of the logs against the normal distribution

seems like a good idea.

(b) temp is a treatment, so needs no explanation

(c) temp and sugar0 are available for use in forming a model of sugar1. When there is one variable to be explained, and there are N variables we could use in the explanation, there are 2^N sets of explanatory variables we might try. Here N=2, and it makes sense to try all 4 kinds of model:

(c1) sugar1 alone, as for sugar0

(c2) sugar1 -vs- temp

(c3) sugar1 -vs- sugar0

(c4) sugar1 -vs- sugar0 and temp

Since sugar1 is continuous and temp has five levels, a one way analysis of variance on log(sugar1) with temp as the factor is suggested. In this case it is actually a silly thing to do, because sugar1 is a fraction of sugar0, and so obviously depends on it. To block this model, we need another rule

[Rule 3] if V1 measures some property of an object, and V2 measures a property of part of that object, V1 must be used in any model for V2

This leaves (c3) and (c4) as sensible models to try.

There is a subtle point: it is *always* a sensible thing to display each observed vari-

able. So a good statistician would also do (c1). But *not* in the expectation of seeing a simple model. Instead, the statistician would want that plot to help him look for mistakes. There is a distinction between tables and graphs designed to make a model apparent, and *diagnostic* tables and graphs designed to show up problems with a model. Here I am not concerned with diagnostics.

So from considering what order the variables were measured in, and which variables are measurements, we've come up with four models to think about, (a), (c1), (c3), and (c4). These weak models can be refined into forms such as (a1-a3) by using our knowledge of the types of measurements. I've not bothered working all the way through this example. The reason is that even with the space of weak models pruned from 12 (analogues of (c1-c4) with each of the three variables on the left) to 4 (it only makes sense to look for an explanation of sugar1) we end up with a lot of things to try for each weak model.

If we were to work out all the plots that would be useful, there would be at least 24 for this example. (Raw and transformed data for each of the weak models, plus raw and transformed residuals from the fitted models.) Now a good statistician might well ask for and look at all of them, and what is more he'd understand all of them. But ASA was originally intended for non-statisticians. 24 graphs for what looks like such a simple problem really is *far* too much for a naive client to cope with, at any rate without a lot of hand-holding from the program explaining why each graph is useful and what the client should look for in each. At the very least, a program which naive clients really could use should keep track of what kinds of graphs and analyses the client can make sense of, and should avoid recommending other kinds of analyses until the client can handle them.

To use Robert Glass's term, ASA is "a success that failed". That is, the statistical reasoning is surprisingly easy. The classification of value spaces presented in chapter 2, the further classification of variables presented in this chapter, and the algebra of experiments presented in this chapter, are indeed adequate for presenting the statistically relevant information about an experiment. It is easy to find and formulate rules about statistical methods and computations, and the planning task is straightforward. The failure is that there is an *enormous* man/machine interface problem to be solved before a consultant which is truly usable by naive clients can be written. Fortunately, solving that problem fell completely outside the scope of this research. The goal of my research was to investigate the representational and inference needs of statistical reasoning, and to find efficient data structures and algorithms for such representation and inference, and that was the "success" part.

Probably all that a client presenting this experiment would really be interested in (or able to use) would be the average after/before ratios for each of the five temperature levels, and an indication of whether they are significantly different from each other. The ideal display

for this would be a "box and whisker" plot[5] of the log before/after ratios side by side for each of the five levels, with a total box-and-whisker as well. 1 is much less than 24, and the client could probably cope with it. Without a client model, it is unclear how to filter out the other plots, as they do make sense and could very well be useful.

A really usable Statistics Advisor would also have elements of an "Intelligent Tutor". For example, the way to determine whether a client could profit from a particular kind of plot is not to ask him whether he knows what it means, but to show him some examples of such plots and ask questions designed to test whether he can see the significant features in the plot. The client should also have the option of entering a teaching mode where the use of such plots is explained.

What is described above is the generation, in an "exploratory" manner, of weak models. But knowing that one variable measures a certain property before treatment and another measures it after treatment is useful for some "confirmatory" analyses as well. One wants to know which variable to plug into the "before" slot of the method and which to plug into the "after" slot, and the temporal order of the measurements tells us this.

Returning to independence, it is possible for a treatment to be *neither* before **nor** after a certain measurement. Consider the following experiment:

J. A random sample of cans of paint is obtained. The density of the paint (V0) is measured for each can. Then the contents of each can are divided in two, the first half being mixed with a randomly chosen type of thinner (recorded as T1) and the viscosity measured (V1), and the second half being painted on a board and exposed for a randomly chosen length of time (T2) and the water permeability of the surface being measured (V2).

Now it makes sense to include V0 and T1 in a model for V1, but it does not make sense to include T2 (as it does not precede V1). Including V2 in such a model would be a bit dodgy as it depends on a treatment not preceding V1. Yet T2 does not follow V1 either. Can we include V2 in a model for V1 or not? Clearly, V2 does give us some information about the paint which might be relevant to V1. But it is heavily contaminated by T2.

This example shows that one rule for excluding a variable V2 from a model for another variable V1 has to be

[Rule 4] if the measurement of V2 is preceded by a treatment T which does not precede the measurement of V1, V2 may not be included in a model for V1.

---

[5]see [Mosteller 78] for an explanation of this term

3.3. How to Develop a Notation for Experiments

Experiments are planned activities. So we look to planning and to computer programming for something to imitate. Graphical notations are common in both areas, and the graphs are generally some sort of flow chart.

The key to my approach here is a metaphor. I call it the "little man" or "little patient" metaphor.

Imagine a little man entering a hospital. He is given a piece of paper, on which his medical history is recorded. He is then shunted from room to room, and at each step something new is added to his piece of paper. Sometimes he is sent to "any available doctor", and the name of the doctor is written on his piece of paper. Sometimes bits of him (in bottles, with his name and the date attached) are sent to various rooms, and back come other pieces of paper which are affixed to the main one.

In this metaphor, the little man represents a **recording unit**[6]. The hospital's computer sees only his card, just as a statistical package such as GLIM [GLIM 78] or SPSS [SPSS 75] sees only the cards that make up (the representation of) a case. A room in the hospital represents a step in the experiment. The X-ray machines and pathology lab represent measurements, and the operation theatres and wards represent treatments. When the little man is assigned to any available doctor, that represents random assignment. A biopsy, or a blood sample, which has its own little piece of paper and undergoes processing which has no effect on the little man himself nor is affected by his experiences after the sample is taken, that represents a **treatment unit**, and reflects the fact that a treatment unit (the thing to which a treatment is applied) can be part of a larger unit.

One more refinement: Consider the Accidents department of the hospital. People do not arrive in it one at a time. They can arrive in car-loads, in bus-loads, or even by the factory-full. Yet each of the people in such a group gets his own medical card. This represents the fact that **sampling units** (the things that are selected) can be groups of recording units. If one member of a group is accepted, all are, and the members of one group tend to be more alike than members of different groups. (A bus-load might all be from the same club, or school. A car-load might all belong to one family. In any case, the people tend to be suffering from the same kinds of problems.)

This metaphor, of a little man in a hospital, is where I started. The distinction between sampling units, recording units, and treatment units is standard in Statistics. It is very important when the purpose of the experiment is to estimate characteristics of a population

---

[6]The terms "recording unit", "sampling unit", and "treatment unit" are standard statistical terminology

rather than checking whether two treatments are different or just fossicking around for anything interesting that might turn up. But the metaphor makes vivid another distinction:

(i) what *kind* of thing is coming into this room? (bus-load of injured people, one man, a biopsy, a urine specimen, ...)

-vs-

(ii) what has *happened* to this thing so far in this experiment? (what is written on the little man's card so far?)

Generally speaking, the same kind of thing comes out of a room as went into it (this is the case by definition in my notation), but something has happened to it in the meantime; we now know more about it than we did before, and as a result of what we have done to it, it may react differently from the way it would have reacted if we'd done something else.

The second factor which influenced the way I decided to describe experiments is formal semantics of computing systems. My semantics for statistical experiments and my semantics for an extension to logic programming [O'Keefe 85] share the same style. In fact the latter followed the former in time.

The third factor which influenced much of the detail is that despite my disbelief in numeric certainty factors, I do believe that it is important for Expert Systems to be able to handle *vague* information. So I saw it as important that ASA should not demand, in consequence of an overly precise notation, information which is not necessary for selecting a method of analysis. If we can select a model entirely on the basis of types, we should do so. If we do not need to know in which order two measurements are made, we should not ask.

This third factor works another way, of course. If we can start with a completely vague "experiment" and refine it as we need to, then it is plausible that the notation could be used for *designing* experiments as well. I have not tried this.

3.4. Signatures

The approach I settled on is that the "processing" part of an experiment is a function from streams to streams, where a stream is an ordered sequence of "units", all of the same type, each with its own history. To keep things simple, and not because I believe this to be the best way to go, a "history" is merely a set of the names of those variables which have been measured or determined to this point. The "type" of the units is a "description", as discussed in Chapter 4.

Examples

Recall that experiment I took a sample of apples, measured their sugar content, stored them at different temperatures, and then measured their sugar content again. An input stream for part of that experiment might be characterised by the following facts in ASA's data base:

```
typical_individual(stream_1, subject_1).      % unit
subject_1 is an apple.                        % type
known(stream_1, sugar0).                      % history
known(stream_1, temp).                        % history
```

The output stream for that part (the measurement of sugar1) might be characterised by the following facts:

```
typical_individual(stream_2, subject_1).      % unit
subject_1 is an apple.                        % type
known(stream_2, sugar0).                      % history
known(stream_2, temp).                        % history
known(stream_2, sugar1).                      % history
```

For the simple experiments I studied, this was adequate. A richer description of histories, perhaps including events which were not measured in any way but which were known to have occurred, may be necessary for some experiments.

Borrowing from semantics of programming languages, every part of an experiment structure is assigned a *signature*, analogous to the type of a function [Gordon 79, Gordon *et al* 79]. This signature describes its input and output.

The signature of part of an experiment is thus a triple

```
(D, B, A)
```

where D is a description of what sort of objects are the "units" for this part of the experiments, B is the set of variables known Before a unit enters this box (this includes measurements pertaining to larger units of which this unit forms a part), and A is the set of variables known After a unit leaves this box. B is necessarily a subset of A.

The signature of a whole experiment is (D, {}, V) where V is the set of all the variable names, which ASA asks for when we start, and D describes the recording units. Note that this does not conflict with my claim that the variables are only known after the experiment is completed: what ASA asks for at the beginning of a consultation is the *names* of the variables, and those *names* are what appear in the set V.

The rest of this chapter describes some particular kinds of steps (boxes, parts of experiments) and notations for them. A step is named by a term

```
<class name>[<parameters>]
```

and the fact that a step has a particular signature is written

<class name>[<parameters>] : <signature>

See the end of section 3.1 for a summary of this notation.

3.5. The Identity step

For each D and V there is an identity function

identity : (D,V,V)

which does nothing at all to the units. This is not a very exciting experiment, but it is sometimes needed as part of a composite treatment.

3.6. The Vague step

For each set O of observations, T of treatments, and V of other variables such that O, T, and V are disjoint, there is a vague "function"

vague[O,T] : (D, V, V ∪ O ∪ T)

which indicates that during this step the treatments T are applied and the observations O are made, somehow. No amount of reasoning or calculation will let us look inside this box; to find out how to refine a vague step we have to ask the client what he did.

3.7. The Observe step

When it does not make any difference which order the observations O are made in, whether because the process of measuring one of them genuinely makes no difference to the other variables, as measuring someone's age does not change his weight, or for whatever reason, we can use

observe[O] : (D, V, V ∪ O)

instead of the corresponding vague[O,{}] step. Strictly speaking, this means rather more than vague[O,{}]. vague[O,{}] means that the variables are measured somehow, but the act of measuring one might influence another, and they might take place in any order, while observe[O] means that we can regard the variables as being measured simultaneously. ASA lacks any rules that care, so it does not actually bother distinguishing between the two. Similarly, ASA doesn't bother distinguishing between "identity" and vague[{},{}].

3.8. Composition of Steps

If step1 : (D, B, X) and step2 : (D, X, A) are steps, then

```
(step1 then step2) : (D, B, A)
```

is a step.

In fact, ASA works this the other way. Suppose ASA has a vague[-,-] step of signature (D,B,A) which needs refining. When it asks the client what kind of step it is, one of the answers he can give is that it is a sequence. For simplicity, ASA only considers a sequence of two steps. It invents a name for each sub-step (step1 and step2, say) and asks the client which variables are determined in the first step. If the answer to that question is the set V, then X is $B \cup V$.

Example.

Suppressing the description argument for now, we now have enough of the notation to describe experiment I.

```
observe[{sugar0}] then
vague[{},{temp}] then
observe[{sugar1}]
```

This leaves us wondering about the treatment, but we can work out what the interesting weak models are with just this much.

3.9. Processing Components in Parallel

This section and the next discuss two ways that components of a unit can be processed in parallel. With the experiment steps described so far, we can represent a sequence of treatments and observations. But consider the following experiment:

K. We are given a sample of households, each of which contains exactly one adult male and one adult female. (Most likely we had a bigger sample and discarded the ones which did not fit this picture.) The man is made to do as many push-ups as he can and the number is recorded. The eyesight of the woman is tested, and recorded.

There are two important things about this: the identity of the components is determined beforehand, and there is no time sequence between the testing of the man and that of the woman. There is no causal flow between them, and it is reasonable to suppose that any relation between the results of these tests is due to the nature and history of the couples before their separate tests. We may regard the experiment steps working on the man and on the woman as *parallel* processes in the sense of computer science, they may be interleaved in any fashion, or performed simultaneously. In order to reap the maximum benefit from this separation, the experimenter should ensure that the man and woman should not communicate until both have been tested, and then both may be tested together subsequently. For if

the woman could sometimes communicate her experiences to the man, there *could* be causal flow from one experiment step to another.

The general picture, then, is that

- the treatment units entering this step of the experiment have components of various different types
- the components exist and the question of which entity corresponds to which component name is settled before the step is begun, typically before the experiment is begun
- the step has several substeps
- the components of the treatment units are routed to the substeps according to which component they are
- only after each substep is complete is the original treatment unit reassembled and subjected to further processing

More formally, suppose that we have k substeps, each of signature

```
substep[j] : (D[j], B, A[j])
```

and that an object described by description D has components C[j] each satisfying description D[j] (it may have other components as well). Then we write down the fact that these substeps are done in parallel as

```
separate[C[1] -> substep[1], ... C[k] -> substep[k]]
```

which has signature

```
(D, B, A[1] ∪ ... ∪ A[k])
```

Pronounce the arrows as "is routed to".

Note that the C[j] are names of components ("field selectors"), not types. We might choose to say just that

```
X is a household ->
    has_one(X,man,M) & M is a human & M is a male &
    has_one(X,woman,F) & F is a human & F is a female.
```

Despite the man and woman components of such a household being given the same description, we can separate them because they are different components.

We might write up this experiment as

```
separate[man -> observe[endurance]
        ,woman -> observe[eyesight]
        ]
```

(Pronounce this as "the household is separated into a man component and a woman component. The man is routed to a stage where endurance is measured and the woman is

routed to a stage where eyesight is measured.") This emphasises that what we have measured is **not** the endurance of the household or the eyesight of the household, but the endurance of the man component of the household and the eyesight of the woman component of the household.

By my conventions, each object has only one component for any given component name, so each substep processes a stream of treatment units containing exactly as many units as the whole step. That is, the substeps substep[j] inherit the sample size from the containing step.

What is the point of representing this kind of structure? After all, we could invent a fictitious sequence and use that. The point is that we come up with different weak models. If we said that we always measured the woman's eyesight and then measured the man's endurance, ASA would never think of using the man's endurance as an explanatory variable in a model for the woman's eyesight. (Unless of course both were measured in the same observe[] step, which is the client's promise that the order of measurement does not matter, indeed his claim that a structure like this underlies the observe[] step.) While if the two are measured independently, then since neither depends on any intervening treatment, both are estimating properties belonging to the same state of the same units, so either variable could sensibly appear in a model for the other.

There is also a difference in interpretation of some tests. Suppose the client indicates an interest in testing whether pushups and eyesight are related. We perform a test of independence. Suppose we measure the woman's eyesight first and the man's endurance second. Then failing to reject the hypothesis of independence means that measuring the woman's eyesight has not been shown to affect the man's endurance. (We have no idea what the man's endurance would have been if the woman's eyesight had not been measured.) If we have processed the components separately, however, failing to reject the null hypothesis means that the woman's eyesight and the man's endurance have not been shown to be associated. This is perhaps a rather tricky point. What if we *do* reject the null hypothesis? In the sequential case, we cannot tell whether we have evidence for an association between the two variables in the population, or whether the association is an artefact of the experiment.

To make this clearer, here is a mechanism that might produce such an artefact. In order to obtain greater relative accuracy, the test of eyesight takes longer as the subject's eyesight is worse. The longer someone's eyesight is tested, the more distressed they become. The more nervous or aggressive someone becomes, the more ready he becomes for fighting, and so the more pushups he can do. The man and woman being allowed to communicate, the man is made nervous or aggressive in proportion to the woman's distress. So the endurance figure for men coming from the same household as a woman of poor eyesight

is higher than it would otherwise have been, while men coming from the same household as women of good eyesight are not affected this way. Clearly, if there was no real connection between the two properties, this experiment will report a negative correlation between eyesight and endurance. But if we test the two people separately, there will be no such effect.

Generally, sequences of measurements are undesirable, and if we can make our measurements separately we prefer to do so.

3.10. Random Assignment

There is another way that we can have components processed in parallel, and that is when a unit has a component which is a set of objects. For example, we normally regard a household as having a set of people as one of its components, as well as things like its address, number of books, number of telephones and so on.

Consider the following experiment:

L. We have a sample of pairs of identical twin calves. For each pair, we flip a coin, and assign one calf to be given hay with Supplement H and the other calf to be given hay with Supplement T. The weight of each calf is measured before and after a week on this feed.

Anticipating the notation presented below, we have

```
observe[weight before] then
assign[twins; feed; H->identity, T->identity] then
observe[weight after]
```

What we have is a stream of treatment units all of the same sort D (here "twin-pair"), each having a component C (here "twins") which is a set of at least k objects (here k=2). All the members of this component set have the same sort Dm (here "calf"). There is a variable V (here "feed"), and k values X[1..k] (here X1=H and X2=T)which this variable may take, and there are k substeps each having the signature

```
substep[j] : (Dm, B ∪ {V}, A[j])
```

where the A[j] are disjoint.

Then

```
assign[C; V; X[j] -> substep[j]]
```

is an experiment step of signature (D, B, A[1] ∪ ... ∪ A[k]). Note that the value k does not appear explicitly in this notation, but is implicit as the cardinality of the range of the variable V.

What is meant by this is that whenever a D unit arrives at this step, we select a random subset of k elements of the set C, and assign the k elements to the k substeps in a

random order. The effect of this is that each of the elements has an equal chance to go in any of the substeps.

We can readily generalise this in various ways to represent factorial experiments. The distinguishing feature of a factorial experiment is that the steps are all essentially the same, differing mainly in the order in which treatments are made. Any particular factorial experiment can in fact be described in the notation of this chapter; the advantage of notations such as that used in [GLIM 78] is that they are compact and irredundant.

There is an obvious equivalence between

```
assign[C; V; X[j] -> step[j] then finally]
```

and

```
assign[C; V; X[j] -> step[j]] then finally
```

The normal form is the latter. Unfortunately, in order to acquire information incrementally, ASA puts the experiment structure in the data base (the implementation is discussed later in this chapter), and so it cannot change the way something has been encoded. I have to rely on asking questions in the right order so that the user enters the normal form. This is unsatisfactory, but has not been a problem with the very simple experiments I've been using as test cases. The commonest case is that all the substeps are the same (though this may imply very different real-world activities!), so the client can be asked if all the substeps are the same, and then instead of refining

```
vague[M,T ∪ {V}]
```

to

```
assign[C; V; X[j] -> ....]
```

ASA can refine it to

```
assign[C; V; X[j] -> identity] then vague[M,T]
```

This is a *very* important structure. The reason is that it introduces a controlled amount of randomness into the experiment so that we are not dependent on random sampling any more. Let's take the example a little further. Suppose we have fitted the model

```
weight_after = K[feed]*weight_before*error
```

where K[feed H] and K[feed T] have been estimated, and we are interested in checking whether the two are different. We do not have to depend on the assumption that we had a random sample of twin calves. We do not have to depend on any assumptions about the distribution of weight in the population of calves. If there were N pairs of twins, we have made N binary random assignments, so there were 2^N possible experiments. If the two kinds of feed do not differ, then the after/before ratios go with the calves, not with the feeds, so we can look at all 2^N possible outcomes and how likely it is that the two feeds would have looked this different just by chance. This is called a *random assignment* method, and there are lots of them. Indeed, in psychology random sampling is almost entirely unknown (so we know much more about the kind of people who volunteer for things than about the other sort) and

random assignment is of extreme practical importance. Nor is random sampling feasible in much agricultural research; an agricultural research station has a fixed set of fields, which in no sense constitutes a random sample of the fields in its nation. Once again, random assignment comes to the rescue.

This section presented random assignment, which is a way of processing similar components of a unit in parallel. The previous section presented separation, which is a way of processing possibly dissimilar components in parallel. In separation, a treatment unit is broken up into components (generally of different types) and these components are processed by substeps. Which substep a component is routed to is completely determined by which component it is. We can tell before the experiment is performed what will be done to a component. In contrast, random assignment breaks up a set of units all of the same type, and there is no possible way of telling before a particular unit is considered which substep will process it.

What the two kinds of substep have in common is that the several components are processed in parallel: if we separate a couple into a man and a woman, or if we randomly assign one twin calf to one treatment and the other to a different treatment, there are two treatment processes in each case proceeding in parallel. Not only in parallel, but there is no communication between the processes. In either case, variables measured in one of the parallel branches may be used in models for variables measured in another of the parallel branches.

3.11. Random Selection

Random selection is like random assignment, except that instead of allocating say 5 rats out of a litter to 5 treatments all happening at the same time, we allocate *one* whole unit to one of a number of treatments.

Consider the following experiment.

M. We have a sample of school children. They are all given the same personality test, then half of them watch a violent film and the other half watch a non-violent film, after which another test is done. The decision as to which child watches what film is made at random.

This is not the same as the previous kind of random assignment. Here units are not broken up, but each unit is routed to one of several possible substeps. There are two ways we could do this. Either each case could be assigned on its own, with the probability that a case would be assigned to each treatment specified (e.g. probability of watching the violent film 0.4, probability of the other 0.6), or we could stipulate the total for each treatment. The latter is generally preferred, as in the former case it could happen that all the subjects

chanced to receive the same treatment. For this reason I have not bothered representing the former possibility.

I represent the latter as

```
select[V; X[j] -> substep[j] | prob[j]]
```

Read this as "a value for variable V is determined at random. The value X[j] is selected with probability prob[j], and units with that value of V are routed to substep[j]." Note that this has no C parameter, as it is not elements of a component which are being routed to the substeps, nor components, but entire units.

The implementation leaves the frequencies vague, as we can generally leave it until the analysis is actually done before we care. In fact each experiment step has a vague frequency associated with it, so the frequency information labels the substeps rather than the arcs to them. What matters is that (a) the assignment of units to steps is random, and (b) each step will have at least one unit routed to it.

This is not as useful as random assignment of components of the same unit, as variables measured in the different substep do not refer to the same objects. However, they do refer to the same *kind* of objects, and to objects from the same *population*. Suppose we have an experiment concerning the feeding of calves containing the following step:

```
select[feed; H->observe[x], T->observe[y]]
```

(Read this as "a value for variable 'feed' is selected at random. Calves with feed=H are routed to a step where 'x' is measured. Calves with feed=T are routed to a step where 'y' is measured.") Now a calf assigned feed=H has not had its y variable measured. We do not know its y property at all, and it might not coincide with any of the measurements we did obtain for y. But we can assume that the *distribution* of the y values is similar in the two groups.

This could be a big problem, the fact that a variable can be measured on one branch of a selection but not another. There are statistical packages which simply cannot handle the case records from such an experiment, not just cheap packages but major, high-priced ones. There is a hack around the problem, and a reason why it is not that much of a problem anyway. The hack is that if a variable y is measured on one branch but not another, then on the other branch it is recorded as "missing". This means that when we look at the relation between y and other variables, a package will only look at the cases for which y was recorded.

The reason why it is not that much of a problem anyway is that the main use of selections is where the substeps differ mainly in the *order* in which they do things. For example,

```
select[first_method;
```

```
          1 -> "teach method A" then "teach method B"
        , 2 -> "teach method B" then "teach method A"
        ]
then "determine method used"
```

Indeed, as here, the different branches often measure no variables at all. Another common case is where the substeps are all identity.

3.12. Dispersing Component Sets

Sometimes we want to look at all the elements of a component, and want to treat them all the same way. For example, suppose we have a sample (random or otherwise) of 21 secondary schools, and assign the school at random to either of two groups, where one group is to use textbook A and the other to use textbook B for a certain form 3 class, and we test all the pupils of each class at the beginning and end of the year.

We have

```
select[textbook; A->identity, B->identity] then
something
```

for the schools, and

```
observe[somevars before] then
wait then
observe[somevars after]
```

for the pupils. It is important to note that while we might have a random sample of schools, we do *not* have a random sample of their pupils, and we cannot even control how many pupils there will be.

Each school actually has some components, one of which is the set of pupils involved in this experiment. We need some way of notionally breaking up the groups so that we can take the pupils as the units. What we want to say is "Here is a stream of schools. Take the "form 3" component of each school, and disperse its components into a common stream of pupils."

Formally, suppose each object satisfying description D has a component C which is a set of elements satisfying description Dm, S is an experiment step of signature (Dm, B $\cup$ {C_id}, A). Then

```
disperse[C; S]
```

is an experiment step of signature (D, B, A). It takes a stream of D-objects (here the schools), takes the C component (here "form 3") of each, forms a stream of Dm-objects (here the pupils) and routes that stream into experiment stage S. So in this example we have

```
select[textbook; A -> identity, B -> identity] then
disperse[pupils; observe[x] then wait then observe[y]]
```

3.12.1. Identifiers

Something to note about this is that we get a new variable, C_id. In this particular example, each pupil gets labelled with the nsame of the school he or she came from. We saw that with the assign[] form as well. But now we discover what it is good for. There are several ways that we might represent the data from an experiment like this. Suppose we have measured one variable on the whole objects, WV, and one variable on the components, CV. We have the variable C_id which exists because this is a dispersal, and every subject comes with a unique subject identifier S_id (which corresponds to the "case number" automatically generated by some packages). One way of representing the experiment in a package is to have a data set with four variables,

```
S_id, WV, C_id, CV  one record per component
```

For example, in SPSS we might have a data set with variables

```
CASENO       (the C_id)
SCHOOLTYPE   (the WV)
SCHOOLNO     (the S_id)
EXAMRESULT   (the CV)
```

This is a fairly wasteful thing to do. If one of the objects has 27 components, its WV value will be recorded 27 times. And of course that is an invitation to mistype one of those replicates. Another way is to have two data sets:

```
S_id, WV            one record per whole object
S_id, C_id, CV      one record per component
```

In GENSTAT[7], we might have

```
'SCALAR' NSCHOOLS
'VARIATE' SCHOOLTYPE(NSCHOOLS)                {S_id -> WV}
'SCALAR NPUPILS
'VARIATE' EXAMRESULT(NPUPILS)                 {C_id -> CV}
'FACTOR' SCHOOLNO(NPUPILS) $ NSCHOOLS         {C_id -> S_id}
```

Readers who know about data base theory will recognise third normal form. See [Maier 83] section 6.2. The reason for recording the C_id is that if we then do a further dispersal on components of the components, we can introduce a third data set

```
S_id, C_id, new_C_id, more_measurements
```

The result of having all these "identifier" variables is that if it is possible for more than one object to go through a given step, those two objects have distinct identifiers, so we can keep their measurements straight. Also, and this is just a practical point, the identifier variables very often indicate the order in which the objects were considered, or have some other ordinal significance, so that plotting some variables against identifiers may be useful in detecting unexpected systematic errors. In the watered-pots example, if we plotted evaporation

---

[7]the syntax has been simplified

against pot number (within watering level), that might reveal a systematic effect due to time of day.

There are three main ways that component identifiers can be obtained.

- own_name
- rank(Attribute)
- serial

The own_name case is when the components come with names of their own. This is the case in the school example. All of the pupils have their own names, which could serve as component identifiers (because all that matters is that no two components of one object have the same C_id). These names do not correspond to any interesting order.

The rank case is when there is a natural ordering on the components, such as age of children in a family, or initial weight of piglets in a litter, and the component identifier is the rank in this order. The Attribute is an attribute of the components, not of the whole objects, and the client should be warned if the substep which examines the components does not measure this attribute properly. If the attribute is measured properly, we can look for relationships between it and the other variables in the usual fashion. If the attribute is not measured properly, we cannot do quite as well. There are two cases here. The first is when the components are dispersed in random order. That is the good one. There are statistical methods which can exploit that. The second is when the components are dispersed in a systematic order related to this attribute. That is the bad one, because we then run the risk of systematic error. If we disperse herds of cows by age of cow, and we are giving them an injection of some drug, and that drug is decomposing as time goes on, then the effect of the drug decomposing will masquerade as an effect of cow age.

The serial case is when the experimenter assigns component identifiers in order of treatment, whether the numbering starts again at 1 for each parent object or distinguishes all the components makes little difference. This is the one we can use to detect unanticipated systematic effects that are correlated with time.

The same classification applies to the top level "case numbers". Even when we have a random sample, it is useful to know whether the cases are studied in random order or whether this order is correlated with some attribute.

So we actually have *three* kinds of variables, not two:

- **treatments**
- **identifiers**
- **observations**

I have never seen this distinction made in a Statistics textbook.

3.13. Filtering

Ideally, the routing decisions in an experiment are random. The benefit of this is that a random decision is guaranteed to be independent of the other variables. But sometimes moral or practical considerations mean that we have to base routing decisions on characteristics of the units. For example, a medical experimenter does not have the option of giving a placebo to a seriously ill patient when there is a treatment which could reasonably be expected to be more successful at curing that particular patient. Less obviously, some treatments may not be possible for some patients. If we want to measure the milk yield of a cow, there is no point in milking the bull!

I call the operation of routing a unit to a treatment based on its already measured characteristics "filtering", because it is most often used to select a subsample for further study. For example, we might test all the cattle in a farm for tuberculosis bacilli in the blood, then filter out the bulls to test for tuberculosis bacilli in the milk of the cows. Such filtering is legitimate and useful. The rejected part of the sample gives us information about the rest of the target population. In this example, we learn two things: the prevalence of tuberculosis in the herd (from the blood test) and the degree to which the milk is affected.

Other uses of the operation are possible but regrettable. An example of such a use would be a medical experiment where patients with severe illness were always routed to the experimental group rather than the control group. This is excellent medical ethics, but it is regrettable Statistics, as it confounds the effect of the treatment with the effect of being severely ill. This operation was included in the catalogue so that ASA could recognise such statistically regrettable experiments and print a warning. None of the experiments which were described to ASA needed this operation.

The notation is based on decision tables.
Let S[i] be steps whose signature is (D, {V} $\cup$ B, A). Each incoming unit is to be routed to one of these steps.

Let T[j] be logical expressions in the variables B. These expressions will decide which of the S[i] a unit is to be routed to.

Let V be a discrete variable with values V[1]...V[k]. This variable will record which of the S[i] a unit was routed to, e.g. a unit routed to S[2] will have the value V[2] assigned to its V variable.

Let R[j] be a subset of {V[1]...V[k]} for each j.

Then

```
when[V; T[j]->R[j]; V[i]->S[i]] : (D,B,A)
```

is a step.

When a subject enters this step, the logical expressions are evaluated. The union of the R[j] corresponding to true T[j] is taken, and one of the values in that set is selected as the value of V. The subject is then routed to the corresponding substep as for 'select'.

This is not a pleasant thing from a statistical view-point, as it means that any difference between the treatments in the substeps is confounded with the difference between the subjects going through those steps. But consider a medical experiment to determine the effectiveness of some new risky technique. We might have

```
observe[severity of disease] then
when[whether_treated;
        severity=high -> {true},
        severity=low -> {false},
        severity=medium -> {true,false};
        true -> Treat, false -> DontTreat] then
observe[degree of recovery]
```

What this means is that if the severity of the disease is observed to be "high", the value of whether_treated will be selected at random from the set {true}, if "low" from {false}, or if "medium" then {true,false}. If whether_treated is selected as "true", the patient will be routed to the step "Treat", while if it is selected as "false" the patient will be routed to the step "DontTreat". So very ill patients will certainly be treated, mildly ill patients will certainly not be treated, and the rest will have a 50-50 chance of treatment. This example should go some way toward explaining why the form is so complex.

This form of step is not included in ASA, as the simple experiments I've looked at have no use for it. Also, I started out with the idea that non-parametric statistics were a Good Thing, but that Statistics was all about significance tests, and that I was entitled to forget about anything that interfered with that. Since then I have learned better. If we are just looking for anything interesting that might turn up, we can ignore the difference between a 'when' step and a 'select' step, provided we remember to warn the user that any "significance levels" that may come out of a package will be meaningless.

A particularly important special case of this is when some class of subjects is discarded:

```
include[T; S] = when[included;
        T -> {true}, not T -> {false};
        true -> S, false -> identity]
```

An experiment where we select a large sample, make a number of tests, and then systematically select a subsample, would be described this way. Some packages would have difficulty handling the data from such an experiment. We would like to represent it as two data sets:

one for the variables measured on all the subjects, and one for the variables measured on the included subjects. In SPSS, for example, we'd have to supply dummy values for the latter variables.

The reason I include this form here, despite its omission from ASA, is to point out that this approach to describing experiments can handle such things naturally.

In actual medical and biological experiments, an important version of this general form is called *censoring*: sometimes a subject just does not live long enough for a measurement to be made. For example, suppose we intend to measure a rat's body weight 2 weeks, 3 weeks, and 4 weeks from the beginning of an experiment. If a rat dies after 22 days, we cannot make the 4 week measurement. The paradigm case of censoring is when the *experiment* does not last long enough. For example, in comparing the merits of 3 different treatments for lung cancer, we might want to record how long each patient lived after treatment was started. But some of the patients (ideally all of them!) may still be alive when the experiment runs out of money. The general nature of censoring is that some event occurs which makes further observation of a subject impossible. This does not fit well into the experiment structures outlined here.

Another form of censoring is when the measuring instruments are incapable of registering the entire range of a variable. For example, the temperature of a patient might be recorded with a thermometer which can only register the range 36 to 38 degrees Celsius. In that case, the reading "38" actually means "38 or greater". This could be represented with the utmost ease by extending the lattice of value spaces with a new generator

```
censored(ValueSpace, LowBound, HighBound)
```

The reason that this was not done for ASA is that ASA doesn't know about any methods for dealing with censored data. Such methods were left out of ASA because they did not appear to add any interesting problems.

3.14. Sampling

A lot of experiments, and a lot of "exploratory" studies, take all the subjects they can get. But other experiments, such as surveys, have a sampling step which selects the subjects for the experiment. The structure of this step is all-important, and the experiment proper is just a single measurement step.

The same kind of approach works for sampling as for treatments. We have a function which is applied to a population and delivers a sample as a result. This is then supplied to the experiment proper, so we have

```
experiment(sample(population)).
```

I have only considered five kinds of sampling procedure, and two of those are no longer represented in the current version of ASA. The five kinds are:

1. exhaustive : all the members of the target population were studied
2. haphazard : a sample was taken, but following no sensible procedure
3. random : what Statistics texts call "simple random sampling with replacement".
4. subsampling : having obtained a sample by some other method, a set component of the subjects is selected, and a simple random sample taken of those components. Each subject thus comes with a pair (or more) of identifiers, naming the object itself and its parent. Subsampling can be nested.
5. stratified sampling : the population is divided into "strata" based on the value of some attribute, then a simple random sample is taken from each stratum. If we have a population of 1000 people containing 900 men and 100 women, a simple random sample of 100 people could contain 93 men and 7 women. A stratified sample would contain 50 men (out of 900) and 50 women (out of 100).

The last two types of sampling have been left out of the current version of ASA since reasoning about sampling does not seem to be interestingly different from the other kinds of reasoning that ASA does, and because rules relating sampling structures and experiment structures are hard to come by. Statistics texts concentrate on one kind of complexity at a time, and this is AI research, not Statistics research. See [Cochran 77] and [Hajek 81] for more about sampling. As a matter of fact, the idea of using a algebra of functions over streams of units came from studying sampling, and was then applied to the body of experiments.

So we have a single entity -- the sampling plan of the experiment -- with its single type. That entity determines a single identifier variable.

3.15. Limitations of this Notation

The experiment structures covered by the classical topic "design of experiments" (see for example [Cochran 57]) can be handled by this notation, but the result is bulky and tedious. The random assignment and selection forms could be augmented with formulas such as GLIM [GLIM 78] supports, and no new theory would be involved.

Most of the difficulties with this notation concern *time*. There are many experiments where the progress of Newtonian time is important, and the present notation cannot handle them. There is an experiment reported in [Velleman 81], for example, where a radio thermometer is attached to a cow, and the temperature is recorded every 10 minutes. Here the "subjects" are not physical objects such as cows, but *events*. There is obviously a very

strong tendency for the temperature at one time to be much the same as it was 10 minutes before. In such experiments, it is the way in which the values at different times are related which is of interest. But this notation is based firmly on the idea that subjects are essentially unrelated.

Nor can the notation handle experiments whose structure changes with time. If an experiment is performed twice, once as a pilot study with few cases and many variables, and once as the main study with many cases and few variables (selected according to the results of the pilot study), We could get by using the "when" form:

```
(common part of experiment) then
include[(sampled in pilot study);
    (part which is only in pilot study)]
```

It would be better if the relationship between the two studies could be explicitly stated, especially as there might be several studies, perhaps a pilot study and slightly different variants of the main study in different regions. The notational approach presented here appears to be adequate for the separate pieces; what is needed is an "analogy mapping" layer on top, so that a Statistics Advisor can see the several studies as instances of a common plan.

3.16. Relationships Between Variables

There are several important relationships between variables which can be deduced from or defined in terms of the structure of the experiment.

An extremely important (and also obvious) relation between two variables is whether or not they are measured on the same units. In fact this relation does not appear explicitly in ASA. ASA rules tend to have the form

```
    ... &
    measured_on(Var1, Sample) &
    measured_on(Var2, Sample) &
    foo(Sample) &
    ...
->  measured_on(Result, Sample) &
    ...
```

where the requirement that the two variables be measured on the same units is stated *implicitly*. If the only benefit of ascertaining the experiment structure was the ability to pose this question, it would still be worth while.

If we were given complete explicit experiment structures to start with, the primitive relation would be that a variable is determined *in* a particular step, and a variable would be measured *on* a particular sample if that sample flowed into a step containing as one of its substeps the place the variable was measured in. But in fact we start with a vague description and refine it only when necessary, so the measured_on facts are primary, and there are rules in ASA to ensure that substeps inherit the measured_on relation.

Another relation is **follows**.

We say that V2 **follows** V1 when every determination of V2 follows in time a determination of V1 for the same subject. "Determination" means either finding out the identifier of something, deciding which level of a treatment to apply, or measuring an observation. The point of "follows" is to track "causality". Clearly, V2 can be influenced by V1 in at least three ways.

- If V1 is a treatment, any causal connection between V1 and V2 can only flow from V1 to V2, simply because that is the way we have constructed the experiment, not because of the variable types or anything else. If we divide 50 people into two groups at random, kick the 25 people in one group in the shins, measure everyone's blood pressure, and find that the kicked group have an average blood pressure 5 points higher than the unkicked group, it is reasonable to postulate a causal connection between being kicked and having an elevated blood pressure. It would not be reasonable to explain being kicked as a result of having a higher blood pressure, because we *know* that our coin-tossing was not affected by the subjects' blood pressure. It would also be silly to look for a common factor which explained both being kicked and blood pressure.
- If V1 and V2 are both observations, things are messier. Even if V2 follows V1, and even if a lot happens in between,
    - V2 could influence V1 but not conversely,
    - V1 could influence V2 but not conversely,
    - they could influence each other,
    - or there could be some common factor which explains any statistical relation between them.

  To continue the example, suppose we took our sample of 50 people and asked a friend to kick any of them he felt like kicking, giving us V1, and then we measured the blood pressure, giving us V2. It could well be that he chose to kick all the fat people! The point of randomising treatments is to eliminate unintended causation of this sort. As an example where causality might plausibly flow either way, consider an experiment where V1 is the total number of cigarettes smoked, and V2 is whether or not someone has cancer of the air tract by age 50. There is no doubt at all that there is a strong link between these two variables. Nowadays it is all but certain that smoking causes cancer. But it used to be possible to argue that maybe people who were cancer prone had a stronger tendency to smoke, and that perhaps smoking might even be beneficial for these people.
- Finally if we know V1 when we measure V2, our expectations may influence the

result. For example, we might anticipate that people whose shins have just been kicked might be feeling aggressive. So we might have their blood pressure measured by a male nurse, and the control group's blood pressure measured by a female nurse. Now we cannot tell whether it is being kicked that raises blood pressure, or having it measured by a male nurse. More subtly, educational experiments are plagued by this effect; the teacher, knowing the results of a preliminary test, cannot help behaving differently to pupils. This may possibly be good pedagogy but it is bad Statistics. Here temporal order does matter; we cannot be influenced by our knowledge of V1 until we have measured it.

A sort of "causality" can flow from the identity of the subject. Your blood pressure is determined by a number of factors, but a lot of them have the same cause, namely your genotype. That is, what you are like depends to some extent on what has happened to you and to some extent on who you are.

Indeed, a branch of Statistics called "randomisation tests" exists, whose basic idea is that if subjects are assigned to treatments at random, we can tell how strongly an effect depends on what has happened to you by pretending that it only depends on who you are, and looking at the distribution of the effect variable over all possible assignments.

Temporal succession is generally irrelevant to identifiers. The constitution of a household, for example, is not determined by our including it in a sample, nor is it likely to be influenced by which brand of shampoo we talk them into using for a month. There are of course experiments where this is not true. For example:

P. We have a random sample of dishes from one day's run of a factory making dishes. They are heated to one of 5 different temperatures and then dropped into a cold bath. Each of the pieces that a dish shatters into is weighed.

In this experiment, a piece is identified by (dish number, piece number within dish). It is entirely plausible that the number of pieces a dish shatters into depends on the temperature to which it has been heated. There are thus two kinds of processes for getting components of an object: those which merely *locate* already existing components, and those which *create* new components. This has nothing to do with variables; it is a question of what we do.

The possibility that a process may create components has unfortunate consequences for knowledge representation. When we describe a new kind of entity to ASA, we can reasonably be expected to list the components that can be located by various kinds of processes. Thus a mammal has a head, eyes, four legs, various other organs, and so forth, and we could be expected to know this in advance, and we could build up a fair bit of knowledge about relationships between these pieces. But a butcher can cut up a beast in many different ways, even just slice it up into 1-inch sections, and we can *not* be expected to

know about all of these ways in advance. If ASA represented actions/processes explicitly, as it has always been my intention to do, we could reasonably expect to be told that such and such a process creates new components so and so.

So the current version of ASA can not handle experiments where components are created rather than located (unless you lie to it). That is not because defining the relation "follows" would be hard, or any of the other reasoning steps, but because ASA does not explicitly represent measurement/treatment processes, and so there is currently no place to put any information we may have about the created components.

That being the case, we can define "follows" this way.

1. No treatment follows anything.
2. No identifier follows anything. (Over-simple.)
3. Every observation follows itself.
4. If identifier I dominates observation O, O follows I.
5. If treatment T temporally precedes observation O, O follows T.

3.17. Some More Examples

This section presents several small but realistic examples to show that experiment structure matters. They do not, of course, show that this notation is "correct:, but they do show how it can be used. I have found that writing down the structure of an experiment in this notation is a useful exercise even when no computer is involved, because it shows up possible nuisance links that I might otherwise have missed. Several text-books which I have used as sources of test cases contain examples which are so ambiguously presented that a unique structure could not be determined; this means that a student working from such an example can easily suppose the structure to be a very different one from that intended by the author, and learn to use the method exhibited on the wrong structure!

3.17.1. Segregated Schools

Suppose we were interested in the question of whether white children did better in segregated schools or in integrated schools.

One approach would be to take a sample of schools and measure for each one (a) whether it was segregated or integrated, and (b) what was the average score of white children on some standard examination. This would be written down as

```
observe[{segregated,score}]
```

There are several sensible models that one might investigate. One of them would be a classical one-way analysis of variance on 'score' with 'segregated' as the class variable. For this model to say anything trustworthy, the schools we studied *must* be a random (or exhaustive) sample from the schools of interest.

Another approach would be to take a sample of uncommitted schools (perhaps schools which had been built but not yet staffed before the study started) and randomly make half of them segregated and the other half integrated, and then measure the scores at the end of the year. This would be written down as

```
select[segregated;
        yes -> identity,
        no  -> identity]
then measure[{score}]
```

Because of the time sequence, there is only one interesting model (try to explain 'score' in terms of 'segregated').

For the two experiments, we would end up with the same calculations. **But the answers would mean different things.** The results from the first experiment tell us what to expect if we look at another school from the existing population which did not happen to be included in the study. The results from the second experiment tell us what to expect if we change things.

The two experiments also have different validity conditions. The first experiment is only meaningful when we have a random sample. When the explanatory variables of a model are not randomly assigned, ASA is obliged to ask the client whether he has a random sample. The second experiment has its own internal randomisation, so ASA does not ask about the sampling plan.

3.17.2. Rat Longevity

Suppose we are interested in testing the hypothesis that eating chemicals which bind free radicals will prolong your life. We might decide to experiment on rats, and to try say four different foods:

```
(a) an unmodified food
(b) a food containing radical-absorber 1
(c) a food containing radical-absorber 2
(d) a food which produces more free radicals
```

We would feed our rats on these foods, and record how long they lived. We are interested in testing the hypotheses that

```
life_span(fed b) > life_span(fed a)
life_span(fed c) > life_span(fed a)
life_span(fed a) > life_span(fed d)
```

One way of conducting the experiment would be to take just-weaned rats and select a food type at random for each rat. This would be written down as

```
select[food_type] then
observe[{life_span}]
```

acting on a stream of rats.

Another approach would be to exploit the fact that rats are born in litters. We would take a sample of litters each containing at least four pups, and assign one pup from each litter to each of the four treatments. This would be written down as

```
assign[pups; food_type;
    X -> observe[{life_span}]]
```

acting on a stream of litters.

In the first experiment, we can compute average life-spans for each of the food types, and look at the differences to see whether they are significant. In the second experiment, we have matched subjects, so we can compute the differences, and look at the averages of the differences. The second experiment is more sensitive. Both experiments have their own internal randomisation, so make sense even when the sample is not randomly selected. (And the sample for the second experiment is not a random sample of litters: litters with 1, 2, or 3 pups will be ignored.)

Unlike the previous example, in this example we perform different calculations, but the results have similar interpretations.

3.17.3. Sibling Rivalry

Suppose we are interested in studying sibling rivalry, and have a sample of 50 brother-sister pairs of much the same age. The idea of the experiment is to give one child in each pair a shiny new toy (carefully selected to be gender-neutral) and the other child a battered version of the same toy, and to look for aggressive behaviour. (Right-thinking children would at once attack the psychologists.) One way of doing this would be to give the new toy to the boy in each pair. This would be written down as

```
separate[boy -> identity          % give new
        ,girl -> identity         % give old
        ] then
observe[{aggression}]
```

Note that the details of the treatment do not show up in this example, because there is no separate "which toy" measurement.

Another approach would be to make the assignment at random, giving the new toy to the boy in some of the pairs, to the girl in the others. This would be written down as

```
assign[the_children; which_toy;
        new -> identity
       ,old -> identity
       ] then
observe[{aggression}]
```

Performing the first experiment would be a profoundly silly thing to do, because we would be completely unable to distinguish the effects of being given the new toy and the effects of being the boy. If all boys tended to be aggressive towards their sisters, we might think we had seen a reaction to the disparity in the toys, when we had only seen "normal" behaviour.

3.18. Summary

Statistical experiments are planned activities. The structure of an experiment is definitely known to a client, and is an important source of information for deciding on an analysis. Methods copied from the study of computational structures can be adapted to the description of experiments.

This chapter presented

- Examples to show that structure is important
- An approach to formalising the description of experiment structure -- the little man metaphor, and its formal analogue of experiment steps as functions over streams
- A specific notation based on this approach
- Examples of the use of the notation

Chapter 2 classified variables according to the values they could take. This chapter introduced another classification:

```
variable
    observation
    treatment
    identifier
        own_name
        serial
        rank_order
            rank_random
            rank_serial
```

Section 3.12.1 mentioned the result that a step can also be read as specifying the structure of a data set as a collection of normalised relations. This is important for generating commands for a statistical package, but is not important for reasoning about which calculations should be done.

Chapter 4
Descriptions

4.1. Introduction

This chapter argues for a new view of types: instead of distinguishing the class of unary predicates as interesting, distinguish functions whose codomain is a complete lattice as interesting. These functions I call *aspects*.

The ABASE knowledge representation scheme, rules aside, turns out to be basically a form of frames, with type hierarchies, multiple inheritance, and a form of defaults. The way it actually developed was that I started from MECHO, and then

- tried to eliminate the use of compound terms {which led to frame-like structures}
- generalised the type system {which led to multiple inheritance}
- rationalised negative rules {which led to lattice-based defaults}
- turned values into types.

Let me make this last point a little bit clearer. Suppose that once upon a time ASA had a predicate

```
        p(Object, Value)
where   function(p(O,V), [O]).
```

(The "function" annotation means that for any given O, there is precisely one true instance of p(O,V). That is, p is a function from O to V.) Now a finite set of distinct values can be turned into a lattice by adding two points: 'unknown' (**1**), and 'inconsistent' (**0**). The type system avoids storing 'inconsistent' type assignments, and all type-like values are assumed 'unknown' initially, so this really is not such a big change. Switching from

```
        p(Object, Value)
```

form to

```
        the p of Object is Value
or      Object is a Value
```

form means that all the mechanisms developed for the type system described in this chapter become available at once. Relational notation is retained for entities which are not known till run time. A particularly important result is that we have the set of possible values available, so that we can show them to the client in a question, do reasoning by exclusion, and so forth.

It is common in AI to divide facts into at least two kinds: ordinary predications, and *types*. Many type systems have been described, ranging from the common "simple type trees" to NETL [Reiter 81, Schmolze 83, Fahlman 79]. These and most other well known type systems have the property that we may regard "X belongs to type T" as a variant notation for "T(X)". That is, types are viewed as unary predicates, and the variations between the types systems concern which sentences of the unary predicate calculus are built into the type machinery.

A simple type-tree is a type system where the types (unary predicates) can be arranged as nodes in a rooted tree, where T1 is an ancestor of T2 in the tree if and only if

```
(∀X) T2(X) ⇒ T1(X)
```

in the logic, and where if T1 and T2 are cousins (that is, neither is an ancestor of the other),

```
(∀X) ~(T1(X) & T2(X))
```

Simple type-trees are easy to understand. They are also very useful. Accordingly, they are very popular. Chapter 7 presents an efficient data structure for simple type-trees and an algorithm for maintaining the data structure as new types are added. Unfortunately, simple type-trees are inadequate for many simple reasoning tasks. Consider the following statements:

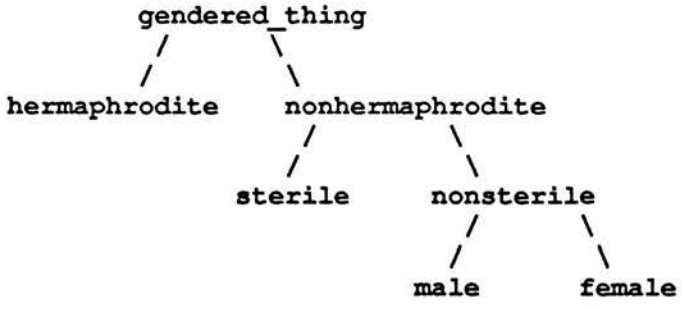
```
[1] an animal can be male, female, sterile, or hermaphrodite
    and it can be neonate, juvenile, adult, senile, or dead
    and it belongs to some species.

[2] an animal and its mother belong to the same species

[3] if an animal is a vertebrate it is male or female
    if an animal is an insect it is male, female, or sterile

[4] there is a classification "languages known" ... ...,
    a human can be classified that way,
    a chimpanzee can be classified that way,
    an animal which is neither a human
        nor a chimpanzee cannot.

[5] if  S is a sample
    and X is a typical_member of S
    and the type of X is T
    then size(S) is a variable
    and the value_space of size(S) is count(T)
```

Simple type-trees cannot handle statement [1]. For example, the combinations (male&juvenile) and (juvenile&crocodile) are compatible but neither entails the other. It is impossible to arrange them in one tree. We can set up gender, age, and species as *separate* simple type-trees, but their combination is not equivalent to any tree. MECHO's type system [Bundy *et al* 76b, Bundy *et al* 77, Bundy *et al* 79b] can handle statement [1], but not the others. The collection of the simple type-trees *does* form a lattice, so the methods of this chapter are applicable.

There are two problems with statement [2]. In MECHO we have the problem of disentangling the "species" part of a type from the rest of the type. This is not an insuperable problem; although the notation currently used provides no way of expressing statement [2], the implementation could support it. The real problem is that while the statement is true for "species" it is *not* true for "variety". The offspring of a cat are cats, but the offspring of a Siamese cat are not always Siamese! This can again be got around by ad hoc means, namely by labelling some of the nodes in the type system as "species" nodes, and generalising the current type of one object up as far as a "species" node before transferring the type to another node.

Being able to hack statement [2] in the implementation is not enough. Statements resembling all of these examples are important in ASA, and it would be much better if the type system were able to express them, not because it had been hacked especially to support them, but because the ability to do so was a consequence of other, more basic, design decisions.

Statement [3] can be handled using more than one type tree, or using a system like MECHO. I have in mind a type tree such as



We can write very simple rules that express statement [3] exactly:

```
vertebrate(X) -> nonsterile(X).
insect(X) -> nonhermaphrodite(X).
```

Despite the fact that its notation would seem to admit such rules, MECHO cannot in fact handle type inferences of this form. That is simply because no need for them was found in MECHO, while such a need *was* found in ASA. There is not the least problem with *stating* type inferences this way. The logic is perfectly clear. MECHO would have no real difficulty in using such rules in a forward chaining way: as soon as we learn that insect(jiminy) we conclude that nonhermaphrodite(jiminy). Indeed, forward chaining on type rules is safe simply because forward chaining in the unary predicate calculus is safe; there are only finitely many propositions which can be formed from the type language.

This brings us to example [5]: for *any* (suitable) "type" T there is to be another type "count(T)". Chapter 2 describes how value spaces are constructed from other value spaces. There is an even simpler example: consider physical Dimensional Analysis. Every measure-

ment in Mechanics has an associated "dimension", which is a triple of rational numbers <M,L,T> denoting the powers to which Mass, Length, and Time are raised. For example,

```
force has dimensions <1,1,-2>
slope has dimensions <0,0,0>
the gravitational constant has dimensions <-1,3,-2>
```

Clearly, there are infinitely many such dimensions. Physical dimensions are an excellent way of dividing measurements into incompatible classes, an Expert System for any sort of mathematical modelling should be able to reason about them. (MECHO relies on its rules being dimensionally correct so that it never introduces ill-typed entities. It doesn't use dimensional analysis to help it select an equation.) But this destroys the finiteness property which makes forward chaining safe. Since such infinite sets of types are so important for mathematical modelling, it is reassuring to know, as a later section of this chapter proves, that a form of finiteness remains, which is sufficient.

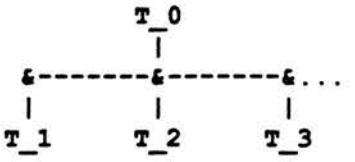
Example [4] shows that even a set of simple type trees may not be enough. The family tree of the primates has humans, chimpanzees, and gorillas at the same level; we want a certain property to be applicable to chimpanzees and to humans but not to gorillas. MECHO's type system does let us have multiple descriptions, but if two nodes in its type tree skeleton both have another classification applicable to them, this other classification must be attached to a common ancestor. In this particular case, we could introduce a dummy common ancestor of humans and chimpanzees, but what if we decided to admit whales to the family of language users?

4.2. A Richer Type System.

Many authors have suggested that simple taxonomies are too restrictive. In this section I examine two alternatives that have been proposed. But these type systems themselves have representational deficiencies, so I propose a new type system which is more flexible.

4.2.1. MECHO

MECHO [Bundy *et al* 79b, Bundy *et al* 79a, Bundy, Byrd & Mellish 82] is an Expert System for solving elementary Mechanics problems. MECHO's types still form a tree, but the MECHO type tree contains *two* sorts of nodes: XOR nodes such as described above, and AND nodes, where



represents a set of axioms

$$(\forall x)\ T\_i(x) \Leftrightarrow T\_0(x)$$

whose point is to allow several different classifications to apply to the same class of entities. For example,

```
object <-> oneof(animate, inanimate).
animate <-> oneof(human, mouse).
animate <-> oneof(male, female).
animate <-> oneof(young, old).
```

says that one sort of a object is an animate object, nothing novel about that, and further says that animate objects can be classified three ways. There is a derivative of MECHO which would insist on separate names for each of the three aspects, e.g.

```
object <-> oneof(animate, inanimate).
animate <-> species.
animate <-> sex.
animate <-> age.
species <-> oneof(human, mouse).
sex <-> oneof(male, female).
age <-> oneof(young, old).
```

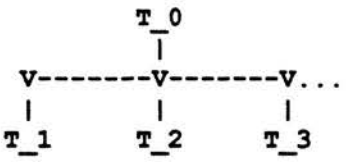
In MECHO the type of an individual is a conjunction of nodes in the type tree, where of course the conjunction must be permitted by the implied axioms. A cunning Prolog coding trick due to Chris Mellish is used to represent these conjunctions. So we can easily say that Hine-Nui-Te-Poo is "human & female & old".

Two nodes in MECHO's tree are compatible iff one of them is an ancestor of the other or their nearest common ancestor is an AND node. Numbering the nodes doesn't slow down finding the nearest common ancestor, but it doesn't make it any faster either. So node numbering doesn't help MECHO.

The representational deficiency is that we can't have nodes *inside* the tree which represent conjunctions, so we can't have a classification which applies only to old humans and not to all humans or all old things. For example, if we have a "Science Fiction" knowledge base which contains both biological and social information, we might want human beings to be classified under Earth-animals, and Fuzzies to be classified under Zarathustran-animals. But we might also want to classify sapient beings as e.g. literate -vs- illiterate. In MECHO the only way to do this is to make "animal" a conjunctive node, with "biological class" and "literacy" as two of its aspects. This has the undesirable side-effect of accepting the question "is this stork literate" as well-typed.

4.2.2. Tangled Hierarchies

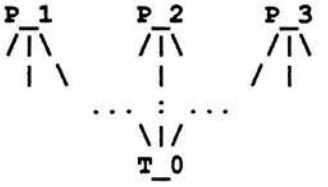
A tangled hierarchy is an acyclic graph, where the nodes are interpreted much the same as in a taxonomy, except that some nodes just represent disjunctions. That is, it is possible to have a node



which represents a set of axioms

```
(∀x) T_1(x) ⇒ T_0(x)
(∀x) T_2(x) ⇒ T_0(x)
(∀x) T_3(x) ⇒ T_0(x)
and no others.
```

A further difference is that a node may have more than one parent. (This immediately wrecks the node numbering method of chapter 7. I have looked in vain for a generalisation which applies to acyclic graphs.) If we continue to interpret arcs as implications,



just means that P_1 & P_2 & P_3 ⇒ T_0. Clearly, if this is to make sense, the nearest common ancestor of each pair of parents must be a simple disjunction.

This gets around the problem mentioned in the previous section. We can say that humans are both Terran-primates and possible-readers, and we can also say that Fuzzies are both Zarathustran-pseudo-primates and possible-readers. And doing so does not prevent us further classifying humans and Fuzzies independently.

But we still have problems. Suppose we have

```
thing <-> terran V zarathustran V literacy.
...
literacy <-> oneof(literate, illiterate).
human -> terran-primate & literacy.
fuzzy -> zarathustran-pseudo-primate & literacy.
literate <-> oneof(only-reads, reads-and-writes).
reads-and-writes <-> oneof(just-scribbles, author).
author <-> oneof(member-of-PEN, common-author).
```

I am sure PEN is not racist in the real world, but in the Science Fiction world we can imagine it excluding Fuzzies. However, even in a tangled hierarchy there is no way to say that the *rest* of the literacy sub-tree is applicable to Fuzzies without implying that *author* is as well.

4.2.3. A Possible Solution.

What we want is a type system with at least the flexibility of tangled hierarchies and as much of the efficiency of simple taxonomies as possible. It seems to me that the root of the problem is the assumption that an individual has a *single* type. This assumption is only made for practical reasons, so if we can treat multiple types efficiently I think we should. Instead of a binary relation

```
Individual isa Taxon,
```

let us consider a ternary relation

```
Individual qua Taxonomy isa Taxon.
```

With such a relation, we can express such facts as

```
fred qua animal isa human.
fred qua voter isa tory.
fred qua located_object isa in_edinburgh.
```

without imposing any purely implementational relation between the "animal", "voter", and "located_object" taxonomies. The type of an individual is defined to be the conjunction of its classifications. Since each taxonomy is a tree, the algorithm of chapter 7 is applicable.

However, just having this ternary relation would permit *any* conjunction, such as a tory cockroach. So in addition to the taxonomies, we need rules to say when an object may be classified under a given taxonomy. These rules could take the form

```
<taxonomy> applies_when <test>.

<test> ::= <taxonomy> isa <taxon>
         | <test> and <test>
         | <test> or <test>

This is to be interpreted as
  (∀X) ((∃Y) X qua <taxonomy> isa Y) ⇔ <test>(X)

where
  (<taxonomy> isa <taxon>)(X) = X qua <taxonomy> isa <taxon>
  (<t1> and <t2>)(X) = <t1>(X) & <t2>(X)
  (<t1> or <t2>)(X) = <t1>(X) | <t2>(X)

For example, we might have

  literacy applies_when
    animal isa human or animal isa fuzzy.
meaning
  (∀X) ((∃Y) : X qua literacy isa Y) ⇔
    (X qua animal isa human) | (X qua animal isa fuzzy)
```

We can take one taxonomy, say "natural_kind", as always applicable.

An important thing to note about these type applicability rules is that they are *monotone*: if we obtain more specific information about an individual the set of taxonomies under which it may be classified does not decrease.

In the remainder of this paper I will generalise the taxonomies from trees to arbitrary complete lattices. They are then called "aspects", and we write

```
the <aspect> of <entity> is <value>
```

rather than

```
<entity> qua <aspect> isa <value>
```

but it is the same underlying idea. The "type applicability rules" of this section are just a special case of description-to-description rules.

4.3. On Being Vague

This section argues that uncertainties attached to predications are ambiguous, and that the uncertainty commonly lies in *some* of the *arguments* of a predication. Descriptions, which can be vague, are offered as an alternative approach.

Expert system tools often provide methods for "uncertain reasoning". MYCIN [Shortliffe 76], with its "certainty factors", and Prospector [Duda et at 78], with its quasi-Bayesian approach, are well known. Shapiro [Shapiro 83] has even shown how the logic programming paradigm can be adapted to uncertain reasoning.

A good deal of criticism has been directed at the rules these and other systems use for combining "certainties". Statisticians urge that the rules of probability should be followed strictly. Proponents of "fuzzy logic" have their panaceas. Yet a major problem with this approach goes unchallenged, and that is the ascription of uncertainties to propositions at all.

Let us consider a statement that might be made to a program like MYCIN.

```
Organism-1 grew in medium-2 (.7)
```

Or, to use a more formal notation,

C[[grew-in(organism-1, medium-2)]] = 0.7

But what does this *mean*? It is clear that we are assenting to the proposition that organism-1 grew in medium-2, with some reservations, but what is it that we doubt? Here are at least some of the possibilities:

- We might be quite certain that something grew in medium 2, but we are not completely certain that it was organism 1. (It might have been another organism floating around the laboratory which happens to resemble organism-1 closely.) This is a form of genuine uncertainty about what happened.
- We might be quite certain that organism 1 grew in something, but we are not completely certain that it was medium 2. (If that glass dish had been placed under a dripping pipe, an unknown quantity of copper salts might have contaminated the medium.) Again, this is a form of genuine uncertainty about what happened.

- We might be quite certain that organism 1 did something in medium 2, but we are not completely certain that whatever it did falls under the heading of "growth". (All of the cells might have become bigger, without any of them dividing.) We are quite sure what we *saw*, what we are not sure about is what to *call* it. This reflects our ignorance, not of events, but of what the other party to the dialogue wants to know.

- We might be quite certain that organism 1 divided and increased its total biomass in medium 2, but we are not completely certain whether the *amount* of growth is *significant*. (After all, one cell out of a thousand dividing in an hour is pretty close to zero.) Again, we are quite sure about what we *saw*. Our uncertainty is a property of the dialogue; we don't know quite what the other party means.

- We may be certain that we were watching organism 1 in medium 2, yet the precision of our instruments may be such that we can only say that the population of organism 1 changed by something between -1% and +5%. This is uncertainty about what actually happened, but we know how accurate our measurements are. Note that if the important fact is whether the organism grew by +30% or more, we can answer "no" without any doubt at all.

So there are at least five different ways that we could be uncertain about this particular statement that organism 1 grew in medium 2.

There are three[8] *kinds* of uncertainty in this example. They are

1. Ambiguity in the dialogue.

2. Possible mis-identification of individuals

3. Imprecise measurements

Uncertainty about whether the thing that grew in medium 2 was organism 1 is a mis-identification problem, as is uncertainty about whether the medium was exactly medium 2. Uncertainty about the numerical value is an imprecise measurement problem. The other problems are ambiguity in the dialogue. In a predicate with more arguments, there are still other ways that these basic types of uncertainty can be manifested. Assigning a single "certainty factor" to a statement as a whole confuses these different sources of uncertainty not only fails to identify which arguments are the introduce the uncertainty, but fails to distinguish the three different kinds. This is a problem because the appropriate response to each kind of uncertainty is different.

We now examine of of these kinds of uncertainty in turn.

---

[8]More kinds can easily be discerned

Ambiguity in the Dialogue

In the context of medical diagnosis, it is reasonable to expect that the client and consultant will share a common language, so that what it means for an organism to "grow" in a medium will be well enough understood by both parties. But when the client and consultant come from different fields, such shared understanding is not as easy to find. Nearly every time I consult a physician, I experience just this sort of confusion. "Do you take any drugs?" he asks, and I wonder "should I mention aspirin?" If asked about my vision, is that with or without glasses? Have you ever tried to describe a head-ache? My limited experience in statistical consulting has been that a lot of time and effort has to be spent building a common vocabulary, and that clients often tend to omit details that they do not consider to be significant, even when specifically asked.

The conversational interface presented to an expert system's client should include means for reducing this kind of uncertainty. At the very least, detailed glosses for all the technical terms should be available. Using one and the same mechanism to say either "I'm not sure I understand the question" and "I understand the question but I'm not sure what the answer is" seems unhelpful.

Misidentification

When an argument of a predicate ranges over entities, it is possible to misidentify the entity. Some common causes of this are

- The entity may be discerned by making imprecise measurements and inferring which entity is responsible. An example might be trying to distinguish the people in a group by their heights. A way of coping with this source of misidentification is to deal as far as possible with the measurements themselves rather than the entity names.
- The entity may be clearly discerned, but the *report* may be subject to transcription error, as when someone measures field 1 but the measurements are read as pertaining to field 7.
- If there is a long interval between observing an entity and recording it, the entity may be misrecalled.

Misidentification is difficult to handle. As ASA deals mostly with abstract individuals (such entities as samples, stages in experiments, measurements &c) misidentification is rare. So I have ignored it. Misidentification is often resposible for gross errors, such as standard "diagnostics" are designed to detect and robust methods [Huber 81, Hampel 86] are designed to cope with.

Imprecise Measurements

Imprecision is another matter. This kind of uncertainty applies to predicate arguments which range over values. When we are talking about an experiment we intend to perform, but have not yet performed, we may not have a definite sample size in mind, we may not be sure how many categories we will divide the range of a continuous variable into, and so on. But even when we do not know whether we will have 25 subjects or 35, we may be perfectly sure that we will have more than 20 and fewer than 50. This is often sufficient precision.

This kind of uncertainty is important in Statistics. We can break it down into two subtypes: *inaccuracy* and *unfaithfulness*.

Inaccuracy applies to measurements. Here "nature" determines a value, over which the experimenter has no direct control, and the experimenter attempts to measure it with more or less precise instruments. Often, knowing which measuring instruments are involved gives us a good idea of the size of error likely. When you are designing an experiment, you can try to cope with inaccuracy by making repeated measurements or by measuring the same value with several different kinds of instruments. (In a questionnaire, one would try to get at the same attitude with several different questions, with scales running in opposite directions, for example.)

Unfaithfulness applies to treatments. If we intend to apply 1cwt/acre of fertilizer to a field, some of it may be spilled or blown away by the wind. Worse still, the significant amount may not be the amount applied but the amount which reaches the roots of the plants in the field, and that may depend on factors which vary all over the field. Often, knowing how the treatment is applied gives us a good idea of the size of error likely. Also, in cases like this, it may be possible to measure effective treatment. For example, one might analyse a sample of the soil from the relevant depth. In a medical experiment to evaluate a drug, one would try to measure the amount of the drug which circulates in the blood or reaches the relevant organ, as well as recording the amount which was administered.

ABASE has no provision for uncertain reasoning, both because I disbelieve in that approach and because ASA has no need of such reasoning. However, ASA *does* need to deal with imprecise figures. It does so using the "descriptions" feature of ABASE. This is a very important point:

> **By using descriptions based on lattices, it is possible to reason with uncertain data in first-order logic, using one mechanism for both types and imprecise values.**

This is not an unexpected bonus of descriptions. The whole description mechanism described in this chapter was deliberately *designed* to serve both purposes.

I use the term "*vagueness*" to refer to the use of lattices to handle imprecise values. For example, I would handle the "growth" example by giving a range:

```
growth_rate(organism_1, medium_2,
    0.1 to 2.0 /* per thousand per hour */).
```

and a rule which needed to use such a fact might be phrased as

```
... &
growth_rate(Organism, Medium, Rate) &
Rate is 0.5 to infinity /* per thousand per hour */ &
...
```

For any lattice, the intervals of that lattice form a lattice. When the base lattice is a total order, as here, the lattice operations on the lattice of intervals are just set union and intersection, though this is not the case in general.

A benefit of this approach is that when the client's data are too vague, an expert system based on descriptions can ask for more information, but by asking a vague question. Consider the previous statement and rule. The two intervals overlap, so the rule might indeed apply, but the interval 0.1 to 2.0 is not wholly contained within the interval 0.5 to infinity, so we do not definitely know that the rule does apply. The intersection of the two intervals is 0.5 to 2.0, so an expert system with this rule and this fact could ask

```
is the growth rate of Organism in Medium
between 0.5 and 2.0 per thousand per hour?
```

The client does not have to supply a specific value of the growth rate in order to answer this question. Clearly, the ability to ask questions which are just specific enough to permit an inference is important.

Oddly enough, dealing with imprecision this way reduces some of the uncertainty which is an artefact of the conversation. Instead of the client wondering how much growth is significant, an expert system based on descriptions *tells* him how much is significant.

4.4. Introducing Descriptions

We have a set E of **entities**. Or rather, a set of names of entities. Entity names are constants. The distinctive characteristic of entities in ABASE is that entity names are not known until consultation time, and that entity names have no significance other than identity and difference:

```
let I be an interpretation function mapping E to
the "real" world (problem domain):
    then I is one-to-one.
```

We have a set D of **descriptions**. Descriptions may be named by arbitrarily complicated terms, and very often the structure of such terms has a lot to do with what they mean. It is up to the knowledge engineer to decide what descriptions look like; this chapter is concerned with formalising descriptions generally, not with prescribing a particular set of lattices. So for the moment, let's remain as vague as we possibly can about what descriptions look like.

The fundamental question is whether a description applies to a particular entity or not. We write

```
e is d
```

to indicate that the description d applies to the entity (named by) e. The question of whether a description applies to an entity (also expressed by saying that the entity *satisfies* the description) has to be settled by means beyond the scope of this work: the question

```
is "john is human" true?
```

is to be settled by checking whether the person who introduced the term "human" would agree that the description he intended by that word does apply to the individual denoted by the word "john". This is exactly like the Tarskian approach to semantics:

```
        "snow is white" is true
iff     "snow" is "white"
```

is/2 is a predicate, so we can write sentences such as

```
e is d1 & e is d2 & ~(e is d3)
```

But a description d is *not* a predicate itself, so

```
d1 & d2 & ~d3
```

doesn't make sense. (Unless the description space D happens to be a Boolean algebra, but that isn't a particularly interesting case.)

Now the first question we have to ask is when two descriptions are to be regarded as equal. There are two equally natural choices: **extension** and **intension**. The intensional view regards identity as part of the definition of the set D, so that if we took

```
D = Z x N          (Z is the integers,
                       N the positive naturals)
E = Q              (Q is the rationals)

x is (y,z) =def x*z = y.
```

then the descriptions (1,1), (2,2), and so forth would all be regarded as different. The extensional view says that two descriptions are to be regarded as equal if they are satisfied by precisely the same set of entities, so that

```
(d1 = d2) =def ((∀e) e is d1 ⇔ e is d2)
```

On the extensional view, (1,1), (2,2), and so forth are all the same description.

There are good arguments for both views. The question is, which is the more useful. In ABASE we are mainly interested in the entities, and not in the descriptions. This suggests that the extensional view is the more useful. We never in fact have occasion in ABASE to test whether two descriptions are the same or not, so I've adopted the extensional view mainly to keep this chapter simple. Descriptions are singled out so that some kinds of reasoning can be done fast, so descriptions with complicated structures where checking extensional equality is expensive are of no interest anyway.

In summary: we have a set of entities E, a set of descriptions D, and a relation "is" between E and D satisfying

$$((\forall e \text{ in } E)\ (e \text{ is } d1) \Leftrightarrow (e \text{ is } d2)) \text{ iff } d1 = d2.$$

ABASE provides you with two ways of making a statement about an entity: descriptors and predications. A predication is a familiar logical atom:

<predicate>(<argument>,...,<argument>).

A descriptor is written as

<entity> is [not] <description>.

What a description looks like is up to the knowledge engineer, though since ABASE is implemented in Prolog, descriptions have to be represented by Prolog terms.

4.5. Comparing Descriptions

After the question "are these two descriptions the same", the next simplest question is "does everything satisfying this description also satisfy that one"? We say that description d1 is *stronger* than description d2, written

$$d1 \leq d2$$

when everything satisfying d1 also satisfies d2. In logical terms:

$$d1 \leq d2 =_{def} (\forall e \text{ in } E)\ e \text{ is } d1 \rightarrow e \text{ is } d2.$$

It follows that (∀d1, d2 in D),

$$(1)\ d1 \leq d1$$
$$(2)\ d1 \leq d2\ \&\ d2 \leq d1 \Rightarrow d1 = d2$$
$$(3)\ d1 \leq d2\ \&\ d2 \leq d3 \Rightarrow d1 \leq d3$$

In other words, (D,≤) is a **partial order**. If we adopted the intensional view, (2) might not hold. To continue the example, (1,1) ≤ (2,2) and (2,2) ≤ (1,1) but (1,1) =/= (2,2).

We define maximally (**1**) and minimally (**0**) vague descriptions:

$$(\forall e)\ e \text{ is } 1 =_{def} \text{true}$$
$$(\forall e)\ e \text{ is } 0 =_{def} \text{false}$$

Since false implies everything and everything implies true, it follows from this definition that

$$(\forall d)\ 0 \leq d$$
$$(\forall d)\ d \leq 1$$

As is usual, we define a "strictly stronger" relation:

$$d1 < d2 =_{def} d1 \leq d2\ \&\ d1 \neq d2$$

That is,

$$(\forall e1)\ e1 \text{ is } d1 \Rightarrow e1 \text{ is } d2$$
$$(\exists e2)\ e2 \text{ is } d2\ \&\ \sim(e2 \text{ is } d1)$$

Suppose there are two descriptions d1, d2 such that

```
d1 < d2
there is no d3 such that d1 < d3 < d2
```

This means that d1 is only just stronger than d2. In the "gender" example, "male" is only just stronger than "nonsterile" which is only just stronger than "nonhermaphrodite". This relation is not transitive; "male" is stronger but not only just stronger than "nonhermaphrodite". It is usual to call this relation **covering** and to say that d2 **covers** d1.

The relation of covering is important when we consider generalisation and specialisation. Suppose that we had been testing whether x is d1, and we have found that this test is too strong. In the absence of any other information about what generalisation to use, we would consider those descriptions d2 which are only just weaker than d1. In the case of a simple type tree, there is only one such d2, and it is the parent of d1 in the tree.

Similarly, if we want to acquire more information about an object from the client, and have no idea what would be useful, we might ask about each description which is only just stronger than the current description. In a simple type tree these are the children of the current node.

4.6. Combining Descriptions

What happens when we are given several descriptions?

```
e is d1.
e is d2.
...
e is dn.
```

We would like to be able to combine all this information into a *single* description. This one description may have different aspects, such as gender, age, species. But that is not having multiple descriptions; the set of aspects is fixed when the knowledge base is constructed, and the aspects of an individual are simply components of a single description.

We want "combination" to resemble conjunction (though not necessarily to be identical to it). Clearly, the combination should not depend on the order in which the statements are made. For we have the conjunction

```
(e is d1) & ... & (e is dn)
```

which follows from these statements. Similarly it shouldn't matter how often we are told that e is d1, it should have the same effect as being told once. So what we want is an operation

```
        /\ : 2^D → D
such that
    [c] (∀e) (e is d1) & ... & (e is dn) ⇒
            e is /\{d1,...,dn}
```

[m] /\{d1,...,dn} is the weakest such description.

For different tasks we shall feel free to invent whatever combination rules (/\) seem useful; but these axioms seem to be a minimal requirement of any useful combination rule. You will search this volume in vain for any definition of $\wedge$ or $\vee$. That is the point: you get to pick a description space which suits your problems.

We define $d1 \wedge d2 =_{def} /\{d1,d2\}$.

It follows that

[1] d1 /\ d1 = d1
[2] d1 /\ d2 = d2 /\ d1
[3] d1 /\ (d2 /\ d3) = (d1 /\ d2) /\ d3 = /\{d1,d2,d3}
[4] d1 /\ d2 = d1 $\Leftrightarrow$ d1 $\leq$ d2

Now a set with a commutative associative idempotent operator $\wedge$ is called a **lower semilattice**, and the operation $\wedge$ is called a **meet**. These properties just fall out of our definition; what we are not guaranteed is the *existence* of $\wedge$. If there is a function satisfying [c] and [m], then it enjoys these properties, but there are partial orders where such a function does not exist.

In fact simple type trees, as I've presented them up till now, are a good example of partial orders where there is no $\wedge$ function. What, for example, would female/\hermaphrodite be? That is why we want **0**. If you take a simple type tree and add a new node **0** (the result is of course no longer a tree) you can define

```
/\{d1,...,dn} =
    let dx be the deepest node among d1,...,dn
    if every di is an ancestor of dx then dx else 0
```

and this is a satisfactory meet. The root of the type tree corresponds to **1**.

One question remains. Should we require that the meet be defined on *all* subsets of D, or on just the *finite* subsets? If D is finite, as it is in MECHO and ASA, there isn't any question. But consider the question set of logical terms with alphabetic variants identified, augmented by **0**, where $\wedge$ is unification, yielding **0** if unification fails. The infinite set of terms {X,f(X),f(f(X)),...} does not have a meet in this set. There is a "completion" of this set, discussed in Chapter 4 of [Lloyd 84], in which meets of infinite sets do exist.

It will make the rest of this chapter simpler to require that the meet is defined on *all* subsets of D.

4.7. Joins

It is a standard result of lattice theory that a complete lower semilattice is a complete lattice [Birkhoff 67]. That is, any subset of D has a least upper bound as well as a greatest lower bound. For least upper bounds we use the symbol \/. We have

```
[1] d1 \/ d1 = d1
[2] d1 \/ d2 = d2 \/ d1
[3] d1 \/ (d2 \/ d3) = (d1 \/ d2) \/ d3 = \/{d1,d2,d3}
[4] d1 \/ d2 = d2 ⇔ d1 ≤ d2

[5] d1 \/ (d1 /\ d2) = d1 /\ (d1 \/ d2) = d1
```

For any subset S of D, we can calculate \/S from

```
/\{d in D | (∀s in S) s ≤ d}.
```

That is, the least upper bound of S is the greatest lower bound of all the upper bounds of S.

Now this is a very surprising result. All we asked for was permission to combine any *con*junction of descriptions into one. But we got in addition an analogue of *dis*junction. Note that it is not equivalent to disjunction. It is certainly the case that

```
(e is d1) | (e is d2) ⇒ e is (d1\/d2)
```

For example, if an animal (such as a bee) is known to be either sterile or female, then it is known to be a nonhermaphrodite. But it is not the case that

```
e is (d1\/d2) ⇒ (e is d1) | (e is d2)
```

since an animal known to be a nonhermaphrodite might be a male.

Joins are good for generalising. (See [Plotkin 69, Plotkin 71].) If we only know that d1 is too strong, we have to consider all the descriptions just weaker than d1. But if we know that all of the descriptions d1,d2,...,dn are at least strong enough, then there is a unique strongest generalisation, namely \/{d1..dn}.

4.8. Negated Descriptions

All along we have been using the notation

```
    e is d
or
    the Aspect of Entity is Value
```

A general problem with Horn-clause based systems such as MECHO and ABASE is their weakness in handling negation. We would like to be able to say

```
    e is not d
or
    the Aspect of Entity is not Value
```

and we'd like to have it stay true once true, just like the positive version. The surprising thing is that this can be done. The definition is that

```
e is not d  is true when d(e)/\d = 0
            is false when d(e)≤d
            is undetermined otherwise
```

If "e is d" is true, "e is not d" must be false. But when the description of e is still vague, e might or might not be d. We have

		e is not d		
e is d		True	False	Unknown
True		impossible	possible	impossible
False		possible	impossible	impossible
Unknown		impossible	impossible	possible

"e is d" and "e is not d" have similar characteristics as goals. The knowledge base is initially consistent with either, and as more information is acquired about e, eventually one of them may become true, whereupon the other will become false. When we are examining a rule body deciding which conjunct to try next, a goal of this kind might be false, in which case we can discard the rule, or true, in which case we can use the fact at once; or it might not be in a definite state, in which case some other conjunct should be chosen.

To prove that

```
(e is d) & (e is not d)
```

cannot be true, we expand the definitions:

```
     (e is not d) & (e is d)
⇒  d(e) /\ d = 0 & d(e) ≤ d
⇒  d(e) /\ d = 0 & d(e) /\ d = d(e)
⇒  d(e) = 0
```

But this means that e is inconsistently described, and ABASE never accepts an inconsistent description.

The other thing which needs proving is that once a negative description becomes true, more information (provided ABASE does not reject it as contradictory) cannot make it false. Suppose then that we have

```
aspect_val(Entity, Old) &
Old /\ Value = false
```

and then we are told

```
aspect_val(Entity, More).
```

The conjunction of the values yields

```
aspect_val(Entity, Old/\More)
```

but

```
(Old/\More)/\Value = More/\(Old/\Value)
                   = More/\false = false
```

QED.

So we have

```
the Aspect of Entity is Value ⇒
    ~ (the Aspect of Entity is not Value)

the Aspect of Entity is not Value ⇒
    ~ (the Aspect of Entity is Value)
```

and once either of these descriptions becomes true, it stays true. The only problem is that *neither* of them need be true. When the lattice in question is a simple taxonomy, this can only happen when the Value we are asking about is more specific than the Current value, e.g. if we ask whether a statistical variable is measured on a ratio scale when all we currently know is that it is measured on an ordinal scale. In general there is another possibility, which is that the Current and the desired Value have a "common sub-type". An important example of this is numeric ranges. When asked for her age, the Client might simply have said "over 21", which is the interval 21 to 100 in this example. One of the rules in the DHSS example is (effectively)

```
the age_class of Client is 'prime' if
    the sex of Client is female &
    the age of Client is 18 to 60.
```

21 to 100 and 18 to 60 have 21 to 60 as their greatest lower bound in the lattice of intervals.

I designed ABASE to accept only positive information from the client. The only way the client can tell ABASE that the sex of Client is not female is by telling ABASE that it *is* some specific (but not necessarily completely specific) other thing. This turns out to be *precisely* the "learning from *positive* examples" part of the Focussing algorithm described in chapter 5. (The lattice used here runs in the opposite direction to that used in the Focussing algorithm, but there is a Duality theorem for lattices.) This tells us what ABASE should do if it were to be extended to accept negative information from the client: it should follow the "learning from *negative* examples" part of the Focussing algorithm. Even accepting negative information about a single attribute requires us to keep multiple black marks, keeping track of information such as "it is not true that the Client is a 35-to-47-year-old male striker in part-time education" would be a frightful task. But if we ever find a task which needs this sort of capability, my having based descriptions on lattices means that the solution is uniquely defined; use the Focussing algorithm.

4.9. Rules Involving Descriptions

ABASE lets you write rules in a wide variety of ways, but for the most part this is syntactic camouflage for Horn clauses. Function-free Horn clauses at that. So we end up with three basic forms of rule:

```
hypothesis ->                  (negative rules)
hypothesis -> predication      (inference rules)
```

```
hypothesis -> description    (description rules)
```

The names of the first two types are copied from MECHO. One of the problems I had with the first version of ASA, which I built on top of MECHO, was that MECHO does not support description rules. So it was not possible to say[9]

```
dichotomy(X) <- discrete(X)&values(X,Set)&card(Set,2)
```

where 'discrete' and 'dichotomy' are types. This is not strictly true. You couldn't say it directly. What you could do was to introduce an auxiliary predicate

```
define isadichotomy(var).
%====== ==================
isadichotomy(Var) -> dichotomy(Var).
```

```
and then write
```

```
isadichotomy(X) <- discrete(X) & values(X, Set)
                                & card(Set, 2).
isadichotomy(X) <- dichotomy(X).
```

Then you have to use 'isadichotomy(X)' everywhere that you would have used 'dichotomy(X)', because the problem is not so much the lack of a notation for description rules as the fact that MECHO will never use a rule to determine whether a description is satisfied.

This hack does provide a way of inferring one or two descriptions in a MECHO rule base, but if you want to have more than a few description rules, you find yourself duplicating the type system in the rule base, which sort of misses the point.

So there are two questions. Can we handle description rules efficiently in an inference engine? And does basing descriptions on lattice theory actually buy us anything?

4.9.1. Negative Rules

As discussed earlier in this chapter, ABASE handles product-form descriptions. That is, instead of saying "entity is description" we say "the attribute of e is range" for each of several attributes. Each factor of the product is a complete lattice. But we identify the least elements of the lattices, and say that if any aspect of an entity is **0** (i.e. is inconsistently described) then the whole description of the entity is regarded as **0** (inconsistently described).

If we are told that

```
        the species of felix is cat.
and     the species of felix is dog.
```

```
then we conclude that
```

---

[9]Mecho uses x<-y where ABASE uses y->x

```
the species of felix is 0 (inconsistently described)
```

from which the given statements do in fact follow. ABASE regards this as a **contradiction**, and refuses to store such nonsense.

All the sources of contradictions in ABASE can be summarised by the logical form

```
Cue & Context -> Description
```

Such rules act in a forward-chaining manner: when an instance of the Cue is asserted, the Context is set up as a goal to be proven, and if/when the Context is determined, the Description is asserted. The Cue may be a predication or a description.

There are several surface forms that these rules may take.

- A negative rule is

  ```
  C1 & ... & Cn ->
  ```

  This can be viewed as

  ```
  C1 & (C2 & ... & Cn) -> the f of X is 0
  ```

  where each of the n conjuncts can be regarded as the Cue. (The rule is actually stored only once.) f and X are arbitrary, but ABASE in fact uses the object taxonomy aspect as f and the most frequently occurring variable as X.

- (Conditional) description-to-description rules have this form.

- Some of the annotations ABASE uses have negative force. (Many of them have positive force.) Examples of such annotations are

  ```
  unique(mother(M,C), [M], [C]).
  antisymmetric(bigger(X,Y), X, Y).
  ```

  These are equivalent to negative rules

  ```
  mother(M,C1) & mother(M,C2) & C1≠C2 ->
  bigger(X,Y) & bigger(Y,X) ->
  ```

  MECHO incorporated such annotations as filters in its proof method. When about to attempt a proof of Cue, it would look in the data base to check whether Context was already known. This is an incomplete strategy, and can result in deductions which do not satisfy the rules. ABASE uses *all* the rules with negative force as filters, even ordinary DD rules: "if I assume Cue, is there an instance of Context in the data base which yields a Description which contradicts existing descriptions?" But when a Cue passes this test (known in MECHO as the "silly" test), ABASE *also* sets up the Context as a proper goal, so that a later proof of the Context will discover the contradiction. This strategy retains completeness. The MECHO strategy is complete when there are no rules for any of the conjuncts in the Context, and each of them will be used to cue the same rule when asserted.

So descriptions play an important part in the way ABASE handles negative rules: the *only* way that ABASE notices a contradiction is when some entity becomes inconsistently described.

4.9.2. Basic Theory

The following theorem is the key to description rules in ABASE.

Let D be a lattice.

Let $f:D \to D$ be a monotone increasing function such that $f(d) \leq d$.

Define $f^0(d) = d$, $f^{n+1}(d) = f(f^n(d))$

Clearly, $f^n(d) \leq f^m(d)$ whenever $n \geq m$.

We would like to talk about the fixed points of f, defining

$$f^*(d) = \lim_{n \to \infty} f^n(d).$$

There are two ways we can be sure that these limits exist.

1. **if *every* sequence $d0 \geq d1 \geq d2 \geq \ldots$ has a greatest lower bound, the limits must exist. But then D is a complete lattice.**

2. **if there is a function $n:D \to N$ such that $f(f^{n(d)}(d)) = f^{n(d)}(d)$ (i.e. if each element of D is only a finite number of "steps" away from a fixed point of f) then the limits must exist.**

Since I have chosen to demand that description spaces are complete lattices, 1) is enough. But I will show below that description rules satisfy 2), so the requirement of completeness could be relaxed. For finite description spaces, completeness comes free, so there isn't any point in relaxing it.

Define $Df = \{f^*(d) \mid d \text{ is in } D\} = \{d \text{ in } D \mid f(d) = d\}$

Define $x \wedge_f y = f^*(x \wedge y)$.

Theorem: (Df,$\wedge$f) is a lower semilattice.

Proof:

let d1, d2, d3, ... be elements of Df.

[1] $d1 \wedge_f d1 = f^*(d1 \wedge d1) = f^*(d1) = d1$.

[2] $d1 \wedge_f d2 = f^*(d1 \wedge d2) = f^*(d2 \wedge d1) = d2 \wedge_f d1$.

```
[3] Lemma: let a be in Df, b be in D.
    Then f*(a/\f*(b)) = f*(a/\b).
    Proof.
        /\ is monotone and f* is monotone, so
        f*(b) ≤ b
        a /\ f*(b) ≤ a/\b
        f*(a/\f*(b)) ≤ f*(a/\b)
        f*(a/\b) ≤ f*(a) = a
        f*(a/\b) ≤ f*(b), hence
        f*(a/\b) ≤ a/\f*(b)
        f*(a/\f*(b)) ≤ f*(a/\b) ≤ a/\f*(b)
        applying f* to all three terms,
        f*(a/\f*(b)) ≤ f*(a/\b) ≤ f*(a/\f*(b))
    End Lemma.
    d1 /\f (d2 /\f d3) = f*(d1 /\ f*(d2 /\ d3))
        = f*(d1 /\ (d2 /\ d3))
        = f*((d1 /\ d2) /\ d3)
        = (d1 /\f d2) /\f d3

[4] Define d1 ≤f d2 iff d1 /\f d2 = d1.
    d1 = f*(d1 /\ d2) ≤ d1 /\ d2 ≤ d1.
    Hence if d1 ≤f d2 then d1 ≤ d2.

End of Proof.
```

We can compute in Df by doing comparisons in D, by doing lattice operations in D, and by then applying f^* to the result.

The fact that the fixed points of a complete lattice under a monotone function form a complete lattice is a well known result of lattice theory [Birkhoff 67] The novel point of this section is that under two additional restrictions on f, namely that $f(d) \leq d$ and that all fixed points exist, the fixed points of a lower semilattice form a lower semilattice with the same ordering. So anyone who dislikes the requirement of completeness (e.g. anyone using finite logical terms as descriptions) can do without it.

4.9.3. Description-to-description rules

A description-to-description rule (DD rule for short) has the form

```
(e is d0) & (e is not d1) & ... (e is not di) -> e is d.
```

where $d0 \wedge di \neq \mathbf{0}$ for i=1..n. Clearly, if $d0 \wedge di = \mathbf{0}$ for any i, it follows that (e is d0) -> (e is not di) and that conjunct is superfluous. Any number of (e is d'j) conjunctions can be merged to form the single (e is d0) conjunct, but negated descriptions cannot be so merged. Thus this is the most general form of rule involving only descriptions of a single entity.

Any rule of this form is necessarily logically consistent. The worst that can happen is that $d0 \wedge d = \mathbf{0}$. ABASE will complain about such rules, because it reacts to inconsistent (**0**) descriptions as it reacts to contradictions. You might prefer to think of this as ABASE having an implicit axiom

```
~ (e is 0)
```

If you want to write such a rule, you can write it as a negative rule. This is only a debugging aid, as negative rules whose left hand side is made up of positive and negative descriptions of the same entity are treated internally as DD rules whose right hand side is (e is 0).

For convenience in what follows I will abbreviate a DD rule as

```
(C,P,N) = (d/\d0,d0,{d1,...,dn})
```

where C is the conclusion, and P and N the positive and negative parts of the hypothesis. ABASE does not use this syntactic form. It is just for this section.

Given a finite set of DD rules (Ci,Pi,Ni) for i = 1..n, we define the function f as follows:

```
f(d) = d /\ /\{Ci | d ≤ Pi & d/\Nij = 0}
```

Since we have a finite number of rules, f(d) is well defined even when description spaces are allowed to be incomplete. Clearly f(d) ≤ d. Now suppose d1 ≤ d2. Then

```
          d2 ≤ Pi ⇒ d1 ≤ Pi
and       d2/\Nij = 0 ⇒ d1/\Nij = 0
```

so f(d1) = f(d2)∧d1∧{some Ci's that d1 catches and d2 doesn't} ≤ f(d2). So f is monotone. Hence by the theorem of the previous section, Df is a description space.

So Df is a description space. What good is that? The point of this result is that we can handle DD rules in the description machinery and keep them entirely out of the main inference engine. The main inference engine will only ever see descriptions which are fixed points of f. Whenever we are given new information (e is d2) when we previously believed that (e is d1), we can at once conclude that (e is f^*(d1∧d2)) rather than concluding that (e is d1∧d2) and waiting to fire the DD rules at some later time.

Before getting too happy about this, assuming that D is a complete lattice only means that limits exist, it doesn't mean we can find them! Now if D is finite, clearly f must converge in some fixed number of steps. But suppose D is not finite. Once a DD rule has been applied to a particular entity, there is never any point in applying it again. So the following elementary strategy will compute f*(d) in finite time:

```
set R = {all the rules}
while there is a (Ci,Pi,{Nij}) in R such that
    d ≤ Pi and d/\Nij = 0
    d := d/\Ci
    delete that rule from R
end while
```

In fact we obtain $o(n^2*k)$ time, where n is the number of rules and k is the "difficulty" of testing the hardest rule. Chapter 6 presents a more efficient algorithm. As ASA doesn't have many DD rules, ABASE uses this algorithm.

4.9.4. Conditional DD Rules

We can also have rules whose conclusion is a description, but which is contingent on more general information. Such a rule has the form

```
generalPart(X) & descriptionPart(X) -> X is d
```

where generalPart is any conjunction of positive literals and descriptions of other variables than X.

We can break such a rule into two parts. One part,

```
generalPart(X) -> rule_enabled(X)
```

is an ordinary rule whose conclusion is an ordinary fact. The other part,

```
rule_enabled(X) -> (descriptionPart(X) -> X is d)
```

is a conditional description to description rule.

What is the point of doing this? The point is to simplify dependency maintenance for descriptions. The fact that a particular CDD rule is enabled is an ordinary fact recorded in the data base, and its dependencies can be maintained in the same way as any other ordinary fact. The current description of an entity is recorded in a

```
<aspect>(<entity>, <value>)
```

fact. (Notionally, there is one such fact per entity, but when the description space is a product, it is more efficient to have one such fact per factor of the description space, here <aspect>.) This fact, like any other, has to be justified. The justification doesn't have to mention the DD rules, as they are timeless truths of the domain which will never go away. The current description can only have been computed by applying the DD rules and all the CDD rules which are currently enabled. So the justification does need to mention which of the CDD rules actually fired. To avoid introducing a special kind of "fact name" that means "such and such a general CDD rule was used", we use the "enabled" facts. The justification for a description of an entity e is the set of rule_enabled(e) facts which are true and which were used.

To compute the current description of e, given the current set of rule_enabled(e) facts, we represent rules as

```
(Enabling,Conclusion,Positive,Negatives)
```

where Enabling is the name of the unary rule_enabled predicate (true for DD rules) and the rest is as before, and do

```
R := {(Fi,Ci,Pi,{Nij}) |
        (Ei,Ci,Pi,{Nij}) is a rule and
        Ei(e) is true and
        Fi is the name of the fact Ei(e)}
J := {}
while there is an (Fi,Ci,Pi,{Nij}) in R such that
    d ≤ Pi and d/\Nij = 0
    if Ci < d then
        d := d/\Ci
        J := J ∪ Fi
    end if
    delete that rule from R
end while
J := J \ {the name of "true(_)"}.
```

Since descriptions are so important, and since it is straightforward to keep them up to date, there is an invariant that we want the dependency system to maintain:

The description of any entity e is always the most specific description supported by the enabled CDD rules.

One kind of conditional description rule is so common that it is worth making a special case of it. MECHO demanded that every predicate be declared, and that its argument types should be specified. So, before you can write "collision(X,Y,T)" in a MECHO rule, you have to say e.g.

```
collision(X, Y, T) ->
    particle(X) & particle(Y) & moment(T).
```

which, in the notation of this chapter, would be

```
collision(X, Y, T) ->
    X is a particle & Y is a particle & T is a moment.
```

ABASE follows in MECHO's footsteps, although it is prepared to do without such a rule if you say that the omission was deliberate.

The actual Prolog implementation of CDD rules stores them in the form

```
cdd(X, TestX, Pos, Neg, Conc)
```

where X is a Prolog variable, TestX is 'true' for a DD rule, or r_enabled(X) for a simple CDD rule. For the declaration of "collision" above, we would get three CDD rules, stored as

```
cdd(X, collision(X,Y,T), any, [], particle).
cdd(Y, collision(X,Y,T), any, [], particle).
cdd(T, collision(X,Y,T), any, [], moment).
```

More generally, any rule whose right hand side is a conjunction of descriptions can be broken up into a CDD rule for each entity mentioned on the right hand side, and perhaps an ordinary rule for an rule_enabled fact for each right hand side entity if the left hand side is not simple. For example,

```
p(X,Y) & X is d1 & Y is d2 -> X is d3 & Y is d4
```

can be stored as

```
p(X, Y) & Y is d2 -> r1enabled(X).
p(X, Y) & X is d1 -> r2enabled(Y).
r1enabled(X) -> (X is d1 -> X is d3).
r2enabled(Y) -> (Y is d2 -> Y is d4).
```

4.9.5. Using CDD rules

MECHO used the type information it had about predicates in two ways. When it was considering whether a goal could possibly be true, it examined the types of the arguments, and if they were not compatible with the predicate the goal was rejected. When it concluded that some fact was true, it used the predicate type to assign types to the arguments of the fact. But types never appeared in the hypothesis of any MECHO rule, so MECHO never had to set "is X a d?" as a goal.

Although ABASE shares no code with MECHO, major aspects of its design are derived from MECHO. So ABASE also uses descriptions in these two ways. When we are considering whether a goal is worth pursuing, we check the descriptions of the arguments against the description of the predicate. Suppose the goal is G, then if

```
cdd(X, G, P, N, C) &
X is P &
X is not N &
X is not C
```

has a solution, then the goal is not consistent with the current set of descriptions.

Whenever we conclude a goal, it may enable some CDD rules. So we have to note which entities may need updating. The rule is that if we have concluded G, and

```
cdd(X, G, P, N, C) &
d(X) /\ P =\= 0 &          % X might be P
d(X) > C                   % X might be C but isn't yet
```

then X may need updating.

There is no problem keeping track of the descriptions implied by the facts that we have deduced so far (for other reasons). If that were all, ABASE would have an easy time of it. But descriptions *can* appear in the hypotheses of rules of ABASE, so ABASE *does* have to set "is X a D" as a goal from time to time. In fact the division of information into descriptions and facts in ABASE is strongly reminiscent of the division into attributes and relations in the Entity-Relation data base model.

The Entity-Relation-Attribute approach uses

- entities, which correspond to entities in ABASE
- values, which are numbers, addresses, colour names, &c, and correspond to the *elements* of description spaces in ABASE

- attributes, which relate entities to values
- relations, which relate entities to entities, and may themselves have attributes.

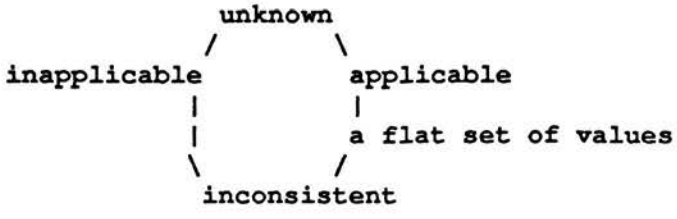
For example, we might have a little data base

```
employee : set of entity (
    attribute name : string
    attribute date-hired : date
    attribute pay : money
)
department : set of entity (
    attribute name : string
    attribute location : address
    attribute sales : money
)
works-in : relation over (employee, department) (
    attribute date-started : date
)
```

We would model this in ABASE by having a description space

```
any
    name
    date
    address
    money
    thing
        employee
        department
        working
```

where name, date, address, and money have the general form



This pentagonal pattern is very common, and is the reason why we cannot base descriptions on modular lattices: no lattice containing this sublattice is modular. There are some other nice properties we can't have either.

In order to say which entities have what attributes, we would have some unconditional DD rules. The abbreviations "X has a D" or "X has no D" stand for "the D of X is applicable" or "the D of X is inapplicable" respectively.

```
X is an employee ->
    X has a name &
    X has a date &
    X has a money &
    X has no address.
```

```
X is a department ->
    X has a name &
    X has no date &
    X has a money &
    X has an address.

X is a working ->
    X has no name &
    X has a date &
    X has no money &
    X has no address.
```

Although I have spelled out in detail which things have and which things do not have each aspect, it is not really necessary to list the aspects which something doesn't have. In any case, these rules can be mechanically generated from a simpler format.

To describe the works-in relation we need a predicate description and an existence rule:

```
works_in(E,D) ->
    E is an employee &
    D is a department.

works_in(E,D,W) ->
    E is an employee &
    D is a department &
    W is a working.

W^works_in(E,D,W) <-> works_in(E,D).
```

The existence rule is to be read as "if we know that works_in(E,D), then we are entitled to create a new entity W to stand for this fact". The three-argument form of works_in is needed so that we have a place to hang the attributes of the two-argument form.

One advantage of the ERA model over the more familiar "flat" relational models is that we can represent the fact that Bill works in the Complaints department but that we aren't sure when. ABASE shares this advantage.

Descriptions subsume attributes. Many of the interesting relations in a knowledge base are relations between entities and values, so one uses descriptions where in MECHO one would have used an ordinary predicate. The reason that MECHO doesn't set up "is X a D?" as a goal is that its simple type system does *not* subsume attributes, and the reason that ABASE does set up "is X a D?" as a goal is that things MECHO would have said with ordinary predicates, ABASE says with descriptions.

The great advantage of lattice-based descriptions to ABASE is that they permit *vagueness*; imprecise information can easily be handled without needing a nonstandard logic. Ordinary facts in ABASE are either true or false, never fuzzy. The notation permits us to be vague about how many legs a particular centipede has without forcing us to be fuzzy about whether Bill works in the Complaints department or not.

So the third use that ABASE makes of CDD rules is backwards chaining. Suppose that we want to show that e satisfies description d, and that this is consistent with the current data base but does not follow from it. Then we find the set of CDD rules which could conclude this (no other kind of rule can conclude a description), and set up their enabling conditions as goals.

```
if cdd(e,G,P,N,C) &
   C ≤ d &                  % e is C implies e is d
   P/\d(e) =\= 0 &          % e might be P
   ~(∃n in N) d(e)≤n
then G is a useful goal.
```

These goals are set up just like other goals. In particular, it may be the case that no rule can help, in which case we have to ask the client whether one of his entities satisfies a description or not. Note that we cannot get away with asking the client about the descriptions of entities that ABASE has invented for its own use, only about entities the client named.

As a general rule, it is better to ask the client about the description of a single entity than to ask about relations between several entities. There are several reasons for this heuristic. One of them is that the question is either a yes/no question "is X a D or not?" or, if we wait until several questions have accumulated, a multiple-choice question "is X (1) a D1, ..., (4) a D4, or (5) something else?". So the amount of information the client has to supply is minimised. Our nett ignorance is decreased by the answer to such a question, whereas if a question involving a relation between several entities were asked the client might be able to name new entities in his answer, which would increase our nett ignorance. This is actually a general heuristic: prefer the question with the fewest variables. Another reason is to try to keep the conversation focussed: we try to pick a question about the same entity as the previous question, so that the user can keep the fewest entities in mind.

4.10. Handling Inheritance Exceptions

One of the reasons we want a type system in the first place is so that we can write down such statements about the world as

```
reptile(X) -> vertebrate(X)
lizard(X) -> reptile(X)
snake(X) -> reptile(X)
amphibian(X) -> animal(X)
mammal(X) -> animal(X)
horse(X) -> mammal(X)
human(X) -> mammal(X)
```

and use this to avoid having to state for each type of animal the number of legs it has by *inheriting* from animal:

```
vertebrate(X) -> leg_count(X, 4)
```

But what about humans (2) and snakes (0)? The general tendency in AI has been to intro-

duce a meta-principle, which says that you chase your way up the inheritance hierarchy and pick the first answer you find. Properties attached at higher nodes are regarded as defaults which can be over-ridden by information attached to lower nodes. Now this doesn't work too well when you have vague information: a marker chasing system such as NETL, told that monty_python is a vertebrate, assumes that this is *all* the information that will ever be available about monty_python, and will happily announce that he definitely has 4 legs, not realising that its information is not sufficient to determine which default should apply. Now in ABASE, thanks to negative type tests, we can explicitly state the rules

```
A:      the species of X is vertebrate and
        the species of X is not snake and
        the species of X is not human
        -> leg_count(X, 4)

B:      the species of X is snake
        -> leg_count(X, 0)

C:      the species of X is human
        -> leg_count(X, 2)
```

Given an explicit lattice, and a set of defaults, we can easily construct these explicit rules, so the burden of listing the exceptions can be taken over by the system, and the knowledge engineer need not bother. The method for doing this is straightforward:

```
for each stated rule "X is D0 -> p(X)"
    locate all the rules "X is Di -> q(X)"
    for which Di ≤ D0 and q(X) & p(X) is inconsistent.

    rewrite the rule as
        "X is D0 & X is not D1 & ... -> p(X)"
```

This method, applied to

```
        function(leg_count(X,N), [X]).

A:      the species of X is vertebrate -> leg_count(X,4)
B:      the species of X is snake      -> leg_count(X,0)
C:      the species of X is human      -> leg_count(X,2)
```

yields the three fuller, consistent, rules stated above.

This process of taking a set of "default" rules and expanding them into ordinary description to description rules with the exceptions explicitly listed is entirely mechanical. The computer can do it, the "knowledge engineer" doesn't have to. Also, it is always possible. We can even add new descriptions to the description space provided that the existing relations in the lattice are not disturbed, and the translation will still be valid. What we cannot do freely is add new "default" rules. It is easy enough to update the existing rules when we add new ones. The trouble is that adding new rules can invalidate existing *conclusions*. For example, if we knew that Tweety was a canary, the rules A,B,C would have told us that Tweety has four legs. (According to "Animal Farm", this is the right answer.) Suppose we decided to add a new rule,

```
D:      X is a bird -> leg_count(X,2)
```

We would then have to repair any rule whose left hand side is more general than "bird" and whose right hand side is incompatible with this one by listing "X is not a bird" as one of the exceptions. But we would be left with an incorrect inference about Tweety. Even if we hadn't known that Tweety was a bird, we might have known that he was a dinosaur (which birds are now classified as being) so rule A would have fired. So the fact that there are no known entities to which the new rule applies is not sufficient to ensure that there are no conclusions which now lack a justification.

There are two possible cures for this problem. One is to prohibit the addition of new "default" rules at run time. That is what ABASE does. This hasn't been a problem in ABASE. The other is to keep track of which DD rules have been used to derive a description, and when a "default" rule is invalidated by a new default rule, to retract all the conclusions which were justified by the old rule and try to rederive them. ABASE exploits the fact that for descriptions it is sufficient to keep track of which facts justify a description and not which rules, regarding the whole type system as a single rule. Since I had no need to add new "default" rules at run time, the simpler cure was appropriate. It *is* important in the Statistics domain to be able to add new taxonomical information at run time, but that poses no problem.

What is the advantage of using rules like this instead of having defaults? The advantage is that if there is insufficient information to decide which rule to apply, *no rule will fire*, and the system will not apply a "default" that it will later have to retract. So, when we learn that monty_python is an animal, none of the three rules A,B,C succeeds and none fails. Then, when we learn that he is a reptile, rule C definitely drops out, but the other two possibilities remain. Finally, when we learn that he is a snake, rule A drops out and rule B fires. Because descriptions are monotone, no further information can disturb this conclusion.

Of course the great advantage of default rules is that in common sense reasoning there are times when you *have* to make a decision based on incomplete information, and the fact that "defaults" handled this way won't jump to conclusions may be a disadvantage. But the great attraction of basing descriptions on lattices rather than unstructured unary predicates is that we do not have to remain completely silent when we have incomplete information. The point is this: we can draw vague conclusions. Let's consider the "legs" example yet again. We introduce a new aspect of an animal: the leg_count. The lattice it is based on is the lattice of intervals of non-negative integers. We write rules

```
A:      the species of X is vertebrate ->
        the leg_count of X is 4 to 4

B:      the species of X is snake ->
        the leg_count of X is 0 to 0

C:      the species of X is human ->
        the leg_count of X is 2 to 2
```

from which we get, as before,

```
A:      the species of X is vertebrate &
        the species of X is not snake &
        the species of X is not human ->
        the leg_count of X is 4 to 4
```

But when the conclusion of these rules is a description, we can do one more thing.

```
    for each stated rule "X is D0 -> X is E0"
        locate all the rules "X is Di -> X is Ei"
        for which Di ≤ D0.

        rewrite the rule as
            "X is D0 -> X is E0\/E1\/..."
```

In this case, we arrive at the *additional* rule

```
A':     the species of X is vertebrate ->
        the leg_count of X is 0 to 4
```

If we are searching for a six-legged creature, A' is enough to tell us that monty_python will not do, as soon as we learn that he is a vertebrate. This is yet another illustration of the fact that vague information is often enough.

We can in fact extend this quite generally: let

```
lhs_i(X) -> X is Ei
```

be a family of rules, and let

```
lhs(X) ⇒  lhs_1(X) | ... | lhs_n(X)
```

be true. Then we can add a rule

```
lhs(X) -> X is (E1\/...\/En)
```

I do not claim that lattice-based inheritance rules are a general replacement for default rules. They are not. Programs which have to deal with the real world cannot always afford to wait until the vagueness has been resolved. But expert consultation programs which can always ask another question if they have to, those programs can benefit from this approach.

This method of handling "defaults" applies only to binary relations between entities and attributes. it does not apply to relations between entities and entities. Consider a rule such as

```
if X is an adult female human being
and Y is a human child
and X and Y are travelling together
and X and Y have the same surname
then X is probably Y's mother
```

The conclusion of this rule is a relation between two entities. As such, the lattice-based approach is completely inapplicable. ABASE has a simple mechanism for handling this kind of default rule as well. The implication arrow is written "-?->" instead of "->" to indicate that the rule is not certain, so we would write

```
        X is a human & X is female & X is adult &
        Y is a human & Y is a child &
        travelling_together(X, Y) &
        surname(X, S) &
        surname(Y, S)
-?->    mother(X, Y)
```

MECHO also has such rules. The way MECHO handles them is to try to solve its goals without using default rules, but if it can't, then to try the default rules. The only real difference is this reluctance to use defaults; once MECHO has applied a default rule the conclusion is regarded as being completely certain. The real use of this is for a form of negation: you might have a default rule that two particles are not connected which fires unless you can prove that they are connected. ABASE does something different. Default rules are used just like ordinary inference rules. When a default rule fires, the same data are recorded as when an ordinary rule fires, so that the explanation subsystem will be able to say where the conclusion came from, but the justification is called an "excuse" rather than a "justification" so that when the dependency subsystem wants to knock out the conclusion of a default rule, it can do so even when the hypotheses of the rule are still believed. The conclusions of default rules are thus regarded as unjustified.

So ABASE has two mechanisms for handling two kinds of defaults. Defaults concerning relations between entities have to be handled in the conventional way, and the dependency subsystem is needed so that default conclusions can be retracted. (It is mainly needed so that the client can retract his answers.) Defaults concerning relations between entities and attributes (qualities, types, numbers, &c) are handled by a lattice-based mechanism which draws conclusions only when they are safe.

A recent paper [Brachman 85] criticises the approach to frames and inheritance found in many present systems. In that paper, Brachman says

> Q: What's big and grey, has a trunk, and lives in the trees?
>
> A: An elephant -- I lied about the trees.
>
> ... some common forms of representation in AI give us the uniform ability to "lie about the trees". These forms then force us to interpret all properties as default properties -- a regime, as it turns out, under which it is not possible to represent genuine universal truths. ... without some sort of definitional force, even the simplest composite descriptions cannot be formed -- and every description in the language is doomed to be a primitive. ...
>
> Given the default interpretation of frames, and the realisation that ISA links do not represent even the simple contingent universals that they seem to, one would suspect that even stronger statements like honest-to-goodness definitions are totally out of the question in standard frame systems. One might also suspect that that's no big deal. Well, the first suspicion is right on the money; but the second is dead wrong. ...
>
> In fact, it is strongly believed that *no* combination of properties is sufficient to capture what it means to be an elephant -- in other words "natural kind" concepts cannot be defined. In contrast, there is nothing more to the story of quadrilaterals than four-sidedness on top of "polygonicity". ...

> Thus, we have most AI representation languages strongly favouring the nonmathematical cases. And with good reason: Why worry about definition if, at best, only quadrilaterals and the like can be defined? Well, consider this: Once we have the concept of an elephant -- natural kind, primitive, or whatever -- from it we can construct an indefinite number of composite concepts, each of which is in a relation to ELEPHANT that is surely definitional. For example, the concept of an elephant with three legs -- call our frame for it "ELEPHANT-WITH-THREE-LEGS" -- is a simple composition of two attributes, each of which is necessary, and the pair of which is sufficient. That is, it is impossible to have an elephant with three legs that wasn't an elephant, and it should be impossible for an object that both was an elephant and had (exactly) three legs to fail to fall under ELEPHANT-WITH-THREE-LEGS. ...

There is a great deal more in that paper, and anyone interested in the design of knowledge representation notations should certainly read it. What is of interest here is that the approach to descriptions and defaults presented here is *not* subject to Brachman's criticisms.

One would take a suitable set of "natural kinds" as "object taxonomy" referred to in chapter 2, or as one of the aspects used here. ABASE permits the definition of properties such as three_legged_elephant by means of logical equivalences:

```
three_legged_elephant(X) <->
        X is an elephant &
        the leg_count of X is 3.
```

This is actually stored as the rules

```
three_legged_elephant(X) ->
        X is an elephant.

three_legged_elephant(X) ->
        the leg_count of X is 3.

three_legged_elephant(X) <-
        X is an elephant &
        the leg_count of X is 3.
```

of which the first two are CDD rules and the last is an ordinary Horn clause. It would also be possible to treat definitions where the right hand side is a description as macros. The point here is that such definitions are allowed and *do* express both necessity and sufficiency.

This is still compatible with having defaults which can be over-ridden by more specific descriptions, because the *general* case is not applied until the specific description is known not to be satisfied.

Where my approach is weaker than frame systems is that if we have a DD rule that says that every elephant has 4 legs, there is no way at all of over-riding that for a specific elephant. We can indeed introduce a class of 3-legged elephants, in which case no elephant will be assumed to have 4 legs until he is proven not to have 3. But if we do not have such a class, my approach cannot handle exceptional individuals.

There is a sense in which it is rather strange to say that all dogs have four legs. We

know that *as stated* it simply isn't true. But, barring monsters, a dog which now has three legs once had four, and most "three-legged" dogs actually have three legs and a stump. This suggests to me that an appropriate way of handling defaults may be to distinguish between *two* kinds of exception.

Exceptional Classes.

This covers cases such as snakes, birds, and human beings. Normally, all vertebrates have four legs, no arms, and no wings. Snakes have no legs. Birds and bats have two legs and two wings. Human beings have two legs and two arms.

This kind of defaulting is fundamentally nothing more than a convenience of notation. It can be very important in practice, because stating that each of thousands of species of vertebrates is *not* exceptional is a time-consuming and error-prone task.

Exceptional Individuals.

Once we have a description of an object, there are many deductions about its properties and components which *usually* follow. For example, a piece of granite normally has a very small amount of gold in it. But *this* particular piece may have been gold-plated. A human being normally has two legs. But *this* particular human being may have been in an industrial accident.

This kind of defaulting is very different from class-level exceptions. The distinction is in part between what a thing *is* and what has *happened* to it. If we learn that this animal is a primate, we will expect to find two forward-pointing eyes on it. But if one eye has been lost in an accident, we still don't know whether to expect the other one to have been lost as well.

I suspect that the best way to handle exceptional individuals may be some form of circumscription [McCarthy 80].

4.11. Reiter's Typed Data Bases

The abstract of [Reiter 81] says "A typed first order data base is a set of first order formulae, each quantified variable of which is constrained to range over some type. Formally, a type is simply a distinguished monadic relation, or some Boolean combination of these. Assume that with each data base relation other than the types is associated an integrity constraint which specifies which types of individuals are permitted to fill the argument positions of that relation. The problem addressed in this paper is the detection of violations of these integrity constraints in the case of data base updates with universally quantified formulae."

This is obviously very close indeed to both MECHO and ABASE. All three require that each relation specify a type for each of its arguments. All three essentially regard types

as some sort of distinguished relation, rather than as "sorts" in the algebraic sense: connections between types are logical rather than algebraic. (This is mainly a matter of emphasis.) However, MECHO and ABASE are Expert Systems, while Reiter is considering deductive data bases. It is interesting to see what difference this makes.

Reiter introduces his type system by observing that it "appears to be a universal characteristic of such argument constraints on relations ... that each such constraint is itself either a simple unary relation ... or a Boolean combination of such simple unary relations."

MECHO's type system is based on exactly this observation: there is a set of base types organised in an and/or tree, and there are derived types which are consistent conjunctions of base types. The restriction on the relations between base types (inclusion and exclusion) and the way derived types are specified is for efficiency: there is a lovely representation of expressible types in MECHO as logical terms so that the lattice structure of the types is isomorphic to the lattice structure of their representations.

Reiter, however, is concerned with generality rather than efficiency, so places no restriction on the formulae that relate types. As in MECHO, he splits the data base into two parts: the *type data base* and the rest. "The type data base is where all information about types resides. Formally, we define a type data base to be any finite set of closed first order formulas all of whose predicate signs are base types and which satisfies the ... T-completeness property." I'll discuss the T-completeness property later. "Formally, the TDB is a set of formulae of the monadic predicate calculus. As is well known (Hilbert and Ackermann, 1950, Principles of Mathematical Logic), the monadic predicate calculus is decidable, i.e. there exists an algorithm which determines, for any formula W, whether or not TDB |- W. This must remain true regardless of how the TDB is represented. Henceforth, we shall assume the availability of such a decision procedure for the TDB. An efficient decision procedure for a large and natural class of TDBs is described in (Bishop and Reiter, "On Taxonomies", UCB CS report 1980)." "We are not seriously proposing that ... the TDB be represented as a set of first order formulae. There are far more efficient and perspicuous representations of the same facts."

So far, everything that he says looks extremely relevant to MECHO, and indeed, to this work. I do need some excuse for not using his methods for checking the consistency of updates, apart from the fact that the updates I allow are so simple that simpler methods will suffice.

There are two difficulties. One is that I have already shown that there are advantages to be gained by rejecting the view that a type is a unary predicate. Indeed, my approach permits and easily handles an infinite number of "types", provided that the lattice operations on them are computable.

The other difficulty is his T-completeness requirement:

> **"For each base type t and each constant c,**
> **either TDB |- t(c) or TDB |- ~t(c)."**

That is, whenever there is any constant c anywhere in the data base, we must know *exactly* which simple types it belongs to and which it does not belong to.

The first problem here is that if we start off with a data base about family relationships, with the obvious types, and try to add the assertion father(frank, jim) MECHO and ABASE will deduce human(frank) & male(frank) & human(jim), and will only reject the update if this is inconsistent with earlier assertions, while Reiter's proposed deductive data base would reject the assertion as ill-typed, since it cannot demonstrate that it is well typed.

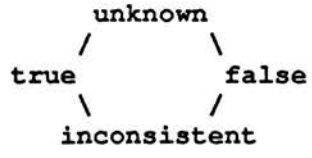
This seems odd. After all, Reiter says "we begin by noting that

> **p(X1,...,Xn) ⇒ t1(X1) &...& tn(Xn)**
>
> **|-**
>
> **p(A1,...,An) ⇔ p(A1,...,An) & t1(A1) &...& tn(An)"**

and defines INT(formula) to be the result of expanding every non-type predicate in the formula this way. He then says "Our approach to data base integrity will be to consider the effects of updating the data base with INT(update)." But he then goes on to show that "an update with INT(C) is equivalent to one in which all literals in INT(C) of the form t(c) have been deleted." (Assuming that it satisfies the integrity constraints.) That is, the types of any constants that appear in an update are checked, and no new information about constants can be derived from an update. Type information about variables in the update is preserved, and this has some interesting consequences.

The result is that before you add a fact to one of Reiter's data base, you may have to separately provide complete information about the constants in the fact to the type data base.

Let's look at ABASE again. ABASE does not regard types as unary predicates. "Type" has been dethroned, and instead of an object having a single type or belonging to a set of boolean types, an object may have any number of attributes, each ranging over a lattice. Now suppose we wanted to use a type system like Rieter's. For each type t(X) we use an attribute t of X ranging over



The description of an object thus belongs to a product of these lattices, where as usual we collapse the "inconsistent" values onto one. This has much the same expressiveness as Reiter's system. We can express a type rule such as (∀X) (male(X) v female(X)) as

```
male of X is false -> female of X is true
female of X is false -> male of X is true
```

or a rule such as (∀X)~(male(X) & female(X)) as

```
male of X is true -> female of X is false
female of X is true -> male of X is false
```

When a formula of the monadic predicate calculus is a natural expression of the relations between your types, using DD rules is very clumsy. The reward you gain by paying this price is that ABASE is perfectly happy with partial information about "types", even if not about anything else.

ABASE shares with MECHO the idea that it is natural to start out in ignorance about an object and to acquire more specific knowledge about it as time goes on. This is the case in any data base for ordinary relations. At one time we don't know who Jim's father is, and then after a consistent update we do know. Why should knowledge about Jim's sex be any different? ABASE goes further than MECHO in exploiting this: one of the ideas in ABASE is that we should leave the description of everything as vague as we can as long as we can, so that we don't demand information from the client that we don't need.

Reiter's requirement is natural enough in a deductive data base. But it would be absurd in an Expert System.

4.12. Summary

This chapter argued that there are several tasks (including ones important to ASA) for which simple type-trees are inadequate. I also suggested that type systems based on unary predicates were not well suited to handling vague information, and that certainty factors attached to predications are not a good way of handling vague information either.

An alternative approach based on complete lattices was presented. This approach lends itself to efficient implementation (see chapters 6 and 7). It provides forms of negation and defaulting.

Most importantly, the lattice-based approach allowed me to treat physical dimensions, statistical variable types, and intervals of numbers, all as descriptions using a uniform mechanism. It was designed to satisfy the needs of ASA, and does so. I believe that any program which reasons about mathematical models will find this approach to descriptions useful.

Chapter 5
The Focussing Algorithm.

It is sometimes said that the bottle-neck in Knowledge Engineering is in Knowledge Acquisition, that is, in the process of getting the rules out of the human expert and into the computer. My experience with ASA has been the direct opposite of this. I have never had the slightest difficulty in acquiring rules from textbooks or examples. The difficulty has been in finding a suitable vocabulary in which to express them, and it is for that reason that so much of this volume describes the way I represent statistical knowledge. The actual rules themselves were very easy to come by. It is worth noting that people using Quinlan's learning algorithm [Quinlan 79] have had much the same experience: 2 months to develop a suitable set of attributes and then 5 minutes for the computer to learn are typical.

Even so, it can be a help if the program can learn. One of the principles adopted in the design of ABASE was that the rule language and type system should demonstrably support learning. It is a sad fact of life that [Plotkin 70] has shown that function-free rule sets are learnable while rules with functions are *not*. This provides yet another motive for ABASE's using function-free Horn clauses as the rule language. Plotkin has also shown that descriptions are learnable, and that algorithm is the topic of this chapter.

The **focussing** algorithm is a learning algorithm described in [Bundy & Silver 82, Bundy *et al* 84, Plummer 83]. The original description is [Young *et al* 77], but that is a 1-page extended abstract. [Young *et al* 77] is a reconstruction of [Winston 75]. [Bundy & Silver 82, Bundy *et al* 84, Plummer 83] do not bring out the fact that the focussing algorithm is based on *lattices*, but describe it for a particular family of lattices, namely products of simple type trees.

Two other papers, [Plotkin 69, Plotkin 71] are useful.

The presentation of the focussing algorithm in this chapter is based on unpublished notes by Gordon Plotkin. The rest of the chapter is original. In particular, I show that when the description space is a product of simple type trees (as [Bundy & Silver 82, Bundy *et al* 84, Plummer 83] assume) a simple *active* learner can identify the intended concept in a logarithmic number of questions, and this is optimal. Active learners are possible in more general lattices, but that algorithm is not as efficient. Perhaps more importantly, I show that the focussing algorithm (and the active learners) can incorporate DD rules.

5.1. The Task of the Focussing Algorithm.

Definition: A **description space** (d-space) is a finite upper semi-lattice with a greatest element: <D,**1**,\/>. That is,

```
Greatest:      (∀x) x \/ 1 = 1
Idempotent:    (∀x) x \/ x = x
Total:         (∀x y) x \/ y is defined
Commutative:   (∀x y) x \/ y = y \/ x
Associative:   (∀x y z) x \/ (y \/ z) = (x \/ y) \/ z

Defn of ≤   :  (∀x y) x ≤ y means (x \/ y) = y
Defn of <   :  (∀x y) x < y means x ≤ y and x ≠ y
Covering:      (∀x y) x covers y means y < x and
                 not (∃z) (y < z < x)
```

Theorem: If $\langle D_1,\mathbf{1}_1,\vee_1\rangle$ and $\langle D_2,\mathbf{1}_2,\vee_2\rangle$ are d-spaces, their product $\langle D_1 \mathbf{x} D_2,(\mathbf{1}_1,\mathbf{1}_2),\vee\rangle$ is a d-space when $\vee$ is defined by

$$(a,b) \vee (c,d) = (a \vee_1 c, b \vee_1 d)$$

This is a standard result, so the proof is omitted.

The focussing algorithm is given a d-space D, and a sequence of examples $(e_1,c_1),\ldots,(e_n,c_n),\ldots$ where each e_i is an element of D and each c_i is either + meaning that the corresponding e_i is a *positive* instance of the concept to be learned or is - meaning that the corresponding e_i is a *negative* instance of the concept to be learned. The task is to find an element E of D such that

$$(\forall x)\ \text{"x is a positive instance"} \Leftrightarrow x \leq E.$$

or to report that the sequence of examples is inconsistent if it can be established that no such element exists.

5.2. The Focussing Algorithm.

The algorithm maintains two markers: a black marker B and a white marker W. The invariant maintained is

$$W \leq E \leq B$$

where W and B are the tightest bounds supported by the examples so far. The idea is that if $e \leq W$ it is definitely in the concept and that if $\sim(e \leq B)$ it is definitely not in the concept. The W and B markers divide the D-space into three partitions:

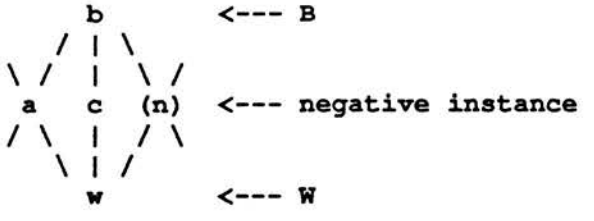
- the White partition = {w in D | $w \leq W$}
- the Black partition = {b in D | $\sim(b \leq B)$}
- the Grey partition = {g in D | $g \leq B$ and $\sim(g \leq W)$}

Clearly, these sets are disjoint and together cover D.

The first example must be a positive instance so that we can initialise the white marker. The algorithm follows *uniquely* from the specification:

```
# initialise #
W := E1
B := 1

# process the rest of the training set #
for i from 2 to infinity while W ≠ B do
    if ci = + then
        # positive instance #
        if ei ≤ B then
            W := W \/ ei
        else
            fail as no E can exist
        fi
    else
        # negative instance #
        if ei ≤ W then
            fail as no E can exist
        else
            B := any maximal element of A where
                A = {b in D | W ≤ b ≤ B & ei ≰ b}
        fi
    fi
od
E := W # = B #
```

Note that this is a non-deterministic algorithm. Suppose we have a configuration such as



Then either a or c would be acceptable as a new value for B. It is only negative instances that create choice points, and then only when B covers more than one d-value ≥ W. If D is such that this cannot happen, the algorithm is deterministic.

Definition: A description space is a **description tree** if it satisfies the unique genealogy constraint:

$$(\forall x\ y\ z)\ x \leq y\ \&\ x \leq z \Rightarrow y \leq z \mid z \leq y$$

In the language of trees, if y and z are both ancestors of x, one of them must be an ancestor of the other. Learning in a single description tree is completely deterministic.

5.3. Learning with a Product of Simple Type Trees

The description in [Bundy & Silver 82, Bundy *et al* 84, Plummer 83] is much more concrete. In those papers, the description space is a product of simple type trees $D = T_1 \mathbf{x} ... \mathbf{x} T_n$, each with its own black and white markers so that $W = (W_1,...,W_n)$ and $B = (B_1,...,B_n)$. As in the previous subsection, a positive example moves all the white markers up. A distinction is made between **near misses** and **far misses**.

Definition: A **near miss** is a negative example $(x_1,...,x_n)$ such that

$$W_i < x_i \leq B_i$$
$$x_j \leq W_j \text{ for } j \neq i$$

That is, all of the features are in the white parts of their description trees except for one grey one. The significance of near misses is that they uniquely determine a black marker to be lowered. We can generalise this definition to lattices: a near miss in a lattice is when the lattice interval [W,B] contains only one place that the black marker can be moved to, i.e. when

$$\wedge \{b \text{ in } [W,B] \mid \neg(e_i \leq b)\} < B$$

Definition: A **far miss** is a negative example $(x_1,...,x_n)$ such that

$$x_j \leq B_j \text{ for } j = 1..n$$

but $(X_1,...,x_n)$ is not a near miss.

That is, all the features lie in the grey or white partitions of their description trees, but at least two of them are grey. The significance of far misses is that they indicate that at least one of the black markers must be lowered, but they do not determine a unique marker to lower. In the general setting, we can have far misses even when there is a single simple feature. In the general lattice setting, a far miss is when

$$\wedge \{b \text{ in } [W,B] \mid no(e_i \leq b)\} = B$$

[Bundy *et al* 84] identify five options for dealing with far misses.

- The **depth first** option is to pick an arbitrary black marker and move it. If later examples are inconsistent with this choice, the depth first option backs up, picks another marker, and restarts. This requires storing all the negative examples (but *not* the positive examples) since the first far miss.
- The **breadth first** option makes *each* possible choice. This requires storing what may be a very large number of different black marker configurations (|D| of them are possible).
- The **teacher** option asks the source of examples which choice to make. This option cannot be used by a program which is conducting its own investigations, and if the teacher and the program have the same representation (as this requires) the teacher might as well supply the concept in one step.

- The **zero** option is to ignore far misses completely and only learn from positive examples and near misses.
- The **avoidance** option is to order the examples so that the problem never arises. It is not clear from [Bundy *et al* 84] whether the authors mean that the teacher is to order the examples or the program.

There is another possibility, and that is to **delay** far misses. This requires storing a set of pairs $\{(i,x_i)|x_i \text{ is grey}\}$ for each far miss. Then, when we move white marker W_k, we do

```
for each grey set G do
    if (k,x_k) is in G and x_k ≤ W_k then
        if |G| = 2 then
            let (j,x_j) be the other element of G
            move B_j down to exclude x_j
            delete G from the grey sets
        else
            replace G by G\{(k,x_k)} in the grey sets
        fi
    fi
od
```

and when we move black marker B_j, we do

```
for each grey set G do
    if (j,x_j) is in G & ~(x_j ≤ B_j) then
        delete G from the grey sets
    fi
od
```

By maintaining a set of grey marks in each tree (a value being marked grey if there is an example in the grey set with that value), we can avoid scanning the grey set except when we know that at least one grey set will change. Note that the delay option does not require us to store complete far misses, just the grey set. As far as I know the delay option is original to me.

With the exception of the zero option, these options are applicable to most nondeterministic algorithms. The delay option is a special case of the general heuristic: postpone decisions as long as possible and when you have to make a choice make the choice with the fewest alternatives.

5.4. The Two Concepts.

At any stage of the learning process, there are two forms of the concept. The most general form of the concept is "it is an instance if none of the features are black". This is the weakest form of the concept permitted by the negative examples. The most specialised form of the concept is "it is an instance if all the features are white". This is the strongest form of the concept permitted by the positive examples. (A rule is strong when it has few instances.)

If one has to use a concept which has not yet been completely learned, whether one uses the weakest form or the strongest form depends on whether one prefers errors of commission (the weakest form may accept non-instances) or omission (the strongest form may reject instances). In the case of an Expert System giving advice in a domain where alternative sources of advice (such as human consultants) are available, it is better for the program to admit that it does not know than for it to give unreliable advice.

A program like ASA should thus use the strongest form of the concept. If we are going to learn rules from a statistician ("method X is relevant to data Y when Y has the following properties") all we really need is the positive examples. Why should we bother with negative examples? There are four reasons.

- The "correct" form of the concept may not be representable in terms of a single bound. If so, the Focussing algorithm will eventually over-generalise. Negative examples are needed to detect this.
- Teachers make mistakes. With both positive and negative examples being supplied, we stand a chance of detecting this.
- A program can actively invent near misses and ask the teacher to classify them.
- In effect, when ABASE accepts information about the type of an object from the client, it is learning a concept from a teacher. Section 4.8 points out that if negative information were supplied (e.g. if the client could answer "no" to a question like "is X a D" rather than "I don't know") we would be in exactly the situation of a learner given positive and negative examples.

5.5. Active Learning

The third reason is worth exploring further. Suppose we have a description space which is a product of description trees: $D = T_1 \mathbf{x} ... \mathbf{x} T_n$. Then an active learner can determine E by asking O(lg(d(D)) questions, where d(D) is the depth of D considered as a lattice.

Definition: an *interval* [L,U] in a lattice is the set $\{x \mid L \leq x \leq U\}$. This is a standard notion.

In a description tree, the unique genealogy constraint has the consequence that

$$(\forall w\ b)\ w \leq b \Rightarrow [w,b] \text{ is a total order}$$

In terms of the tree, [w,b] is w and all its ancestors, excluding the proper ancestors of b, so it is not surprising that this list is a total order. The exciting thing is that this means that we can do binary search.

Definition: for any $w \leq b$,

midpt(w,b) is that x in [w,b] for which
#[x,b] ≤ #[w,x] ≤ #[x,b]+1

This is the analogue of ceil((w+b)/2) in ordinary binary search.

Definition: for any w < b,

pred(b;w) is that x in [w,b] for which b covers x

In tree language, pred(b;w) is the unique son of b which has w as a descendant. This is the analogue of b-1 in ordinary binary search.

We arrive at the following algorithm:

```
get an initial positive example (x_1,...,x_n).
set each W_i to x_i
set each B_i to the root of T_i

while there is an i such that W_i ≠ B_i do
    while W_i ≠ B_i do
        set each y_j to W_j for j ≠ i
        set y_i to midpt(W_i,B_i)
        # (y) is either an example or a near miss #
        if (y_1,...,y_n) is an example then
            set W_i to y_i
        else
            set B_i to pred(y_i,W_i)
        fi
    od
od
```

For each aspect i, the inner loop is performed at most ceil(lg(d(Ti))) times, so the maximum number of questions that can be asked is

```
ceil(lg(d(T1))) + ... + ceil(lg(d(Tn)))
```

which is clearly optimal. The very first positive example converts the problem from searching in a space with #T1x...x#Tn elements to search in a space with at most d(T1)x...xd(Tn) elements, and given that each question yields only one bit of information, the optimality of this algorithm follows from information theory.

Indeed, this approach can be applied in the general setting where the T_i are semilattices and not just trees. Unfortunately, the intervals are no longer total orders, so we do not obtain a binary search algorithm. Instead we obtain

```
get an initial positive example (x_1,...,x_n).
set each W_i to x_i
set each B_i to 1

while there is an i such that W_i ≠ B_i do
    while W_i ≠ B_i do
        set each y_j to W_j for j ≠ i
        Z := W_i
        for each z | z covers W_i and z ≤ B_i do
            y_i := z
            if (y_1,...,y_n) is an example then
                Z := Z \/ z
            fi
        od
```

```
            if Z = W_i then
                B_i := Z
            else
                W_i := Z
            fi
        od
    od
```

The cost of this can be as high as #T1+...+#Tn questions.

The surprising fact is that *for this particular type of concept*, an active learner is simpler than a passive learner, and what is more can proceed deterministically, without ever making any mistakes. This is because we can avoid far misses entirely. Whether this applies to other problems, or whether it is possible only because of the extreme simplicity of the task I know not.

The fact that an active learner can be simpler, and can learn faster than a passive one is surprising. The general attitude in the learning literature seems to be that supervised learning is the right thing to study right now because it is simpler than active learning. The active lattice learner has not, to my knowledge, been described before.

5.6. Lattices.

Chapter 4 argues for product-of-lattices description spaces as a replacement for type hierarchies. While it is true that a lattice is a semilattice, and hence the Focussing algorithm can be used, it may look as though I am asking for more than I need. Why demand that d-spaces should be lattices when semi-lattices (such as type hierarchies) will do? Indeed, one of my arguments for removing the privileged role of types is that we may then be able to replace one hard-to-manipulate DAG by several simple trees. Asking for lattices may seem inconsistent.

The first answer to this question is that we need something that corresponds to conjunction within a single d-space. The join operation V tells us what to conclude if we are told that "fred is an elephant OR fred is a weasel", namely to conclude the upper bound "fred is a mammal". But what are we to do if we are told that "fred is an elephant *and* fred is a weasel"? Or less obviously, "x is divisible by 2 *and* x is divisible by 7"? The meet operation (greatest lower bound) tells us what.

The second answer to this question is that any *finite* semi-lattice is very nearly a lattice already. If D does not have an element **0** which is smaller than all the other elements, adjoin a new element **0** and give it this property. So

$$(\forall x)\ \mathbf{0} \leq x$$

Now we define

$$(\forall x\ y)\ x \wedge y = \bigvee\{z \mid z \leq x \ \&\ z \leq y\}$$

Since there is a least element, the set above is non-empty, and since D is finite, it is finite, so V{z...} is defined. The proof that this definition of Λ satisfies the lattice axioms is straightforward. Furthermore, any finite lattice with a minimum (**0**) and a maximum (**1**) element is necessarily complete, so a finite description space in the sense of this chapter is also a description space in the sense of the previous chapter.

When D was a tree, it turns out that

$$x \wedge y = \text{if } x \leq y \text{ then } x \text{ else if } y \leq x \text{ then } y \text{ else } 0.$$

This means that all the nodes in a tree-like d-space are to be regarded as exclusions, which is exactly the way simple type trees are normally intended.

It is a standard result of lattice theory [Birkhoff 67] that a partial order P can be embedded in a lattice. The lattice is the set of all subsets of P which are closed below. A subset Q is closed below when

$$(\forall x\ y)\ x \text{ in } Q \ \&\ y \leq x \Rightarrow y \text{ in } Q$$

Any lattice which embeds P has a sublattice isomorphic to this one. Applying the focussing algorithm to this derived lattice turns out to be equivalent to maintaining two *sets* of markers W, and B. Given this derivation of the algorithm for learning in a partial order, coding the algorithm is straightforward. In an early draft of this section a couple of years ago, the algorithm was presented in full, because I thought I had found a generalisation of the focussing algorithm. In fact, of course, it is a specialisation. This illustrates the power of lattice theory.

5.6.1. DD rules and Learning

Chapter 4 proved that adding DD rules to a lattice leaves you with a lattice. We can compute in the new description space (the quotient of the base description space under the equivalence "d1 and d2 are rewritten to the same result by the DD rules") easily enough:

```
to compute d1 /\ d2:
    set x = d1 /\ d2 in the base space
    apply DD rules to x until convergence
    result is x'

to compute d1 \/ d2
    apply DD rules to d1 until convergence
    apply DD rules to d2 until convergence
    result is d1' \/ d2'
```

Chapter 6 shows that computing in the quotient description space can be done efficiently.

This goes some way towards solving a problem which the focussing algorithm ap-

peared to have. As described in [Bundy *et al* 84], the focussing algorithm just works from a set of trees, and is capable of coming up with silly answers because it cannot be told about any relations between the types other than those implicit in the trees. But we now see that this is only true of that particular instance of the algorithm.

The result that the focussing algorithm needs no extension at all to handle DD rules (which is to say, most interesting rules relating descriptions) is new. Given chapter 4, it appears obvious, but that is the point: Lattices are Good For You.

This result is briefly stated, but it is very important. Learning algorithms in general have a tendency to find rules which cover the training set but are intrinsically silly. Very simple systems such as Expert-Ease are particularly subject to this failing. To obtain sensible rules, it helps if we can provide rules about the concept description language, saying that some combinations are forced and others are impossible. The significant feature of the focussing algorithm is that because it is based on lattices, there is a useful class of rules which the (abstract) algorithm can *already* handle.

DD rules can be combined with an active learner. Before asking any question, the DD rules are applied to it, so that the question asked is a fixed point. Since each question is a generalisation of the current white marker, the fixed point it is mapped to cannot be below the fixed point the current white marker is mapped to. Hence by induction we can show that no question will ever be mapped to "contradiction". No question will ask about the status of an impossible object.

5.7. Conclusion

The focussing algorithm (or rather, family of algorithms) is a method of learning concepts which can be viewed as elements of a lattice. A wide variety of problems can be cast in this form.

The result that the method can used to derive an active learner which asks relatively few questions and never needs to backtrack appears to be new. A simple active learner can identify a concept in a product of trees in an *optimal* number of questions.

An important new result is that the focussing algorithm, because it is based on lattices, can easily accommodate description-to-description rules. In particular, DD rules can serve as negative constraints (this combination of attributes can never occur -- is mapped to **0**) so that if the learner wanders into an "impossible" region of the description space it can backtrack out instead of failing to converge for lack of counter-examples.

In the Expert System world, I wanted to handle conjunction. In the Learning world, we want to handle disjunction. But in both cases we are naturally led to complete lattices.

Chapter 6
Finite Fixed-Point Problems.

This chapter presents an algorithm for solving a class of fixed-point problems in linear time. This is an important result for this work, because it applies to DD rules. For a given lattice, the total time taken for all updates to the description of an entity is at worst proportional to the total size of the DD rule-set. It is also shown how this algorithm can be applied to forward chaining in sets of function-free Horn clauses.

When specialised to sets of propositional Horn clauses, the algorithm reduces to the well known LINCLOSURE algorithm [Beeri 79]. The fact that models for sets of propositional Horn clauses can be found in linear time has been known since 1979; that is not a new result. However, it too is significant for ABASE. ABASE keeps a record of every inference step. Each such record is a labelled propositional Horn clause:

```
{according to rule so and so}
fact_1 & ... & fact_n -> conclusion
```

This is used for providing explanations, and for belief revision. A client can retract any answer he has given and any information he has volunteered (except additions to the description system). When this happens, all the conclusions which depended on the retracted statements are withdrawn. As LINCLOSURE is used to propagate conclusions along existing paths, the cost of rederiving *all* the conclusions which are still valid is at worst linear in the size of the inference records which were invalidated.

The algorithm described in section 6.2 can be viewed as a variant of LINCLOSURE. Its actual historical development was as a generalisation of a variant of topological sort for hyper-graphs. The fact that this algorithm has a large number of relatives is testimony to the ubiquity of fixed-point problems: it is even possible to compute transitive closures this way.

6.1. The Problem to be Solved.

We are given a set of *variables* {x[1],...,x[n]}.

Each variable x[i] ranges over a *lattice* L[i] having finite depth d[i]. The *depth* of a lattice is the length of the longest chain

$$e_0 < e_1 < e_2 < \ldots < e_{depth}$$

which can be formed from elements of the lattice. Different variables may range over different lattices. All the lattices are required to have a least element (0) and a greatest element (1). Note that any of the lattices may have infinitely many elements; what matters is the length of the longest chain.

The *depth of the problem*, d, is defined as d[1]+...+d[n].

We are given a set of monotone increasing functions

```
f[j] : L[j,1] x ... x L[j,a[j]] -> L[j,0]
```

Definition: a *finite fixed-point problem* is a set of inequalities

$$x[k] \geq e[k]$$

where the right hand side of each inequality is a well-typed expression constructed from the given variables and monotone functions. We are to find a minimal model (or fixed point) of this set of inequalities, where

Definition: a *model* for a set of inequalities is an assignment of lattice elements to variables which makes the inequalities true,

Definition: given that M1 and M2 are models for a set of inequalities, M1 is said to be a *proper submodel* of M2 if

$$(\forall 1 \leq i \leq n)\ M1(x[i]) \leq M2(x[i])$$
$$\&\quad (\exists 1 \leq j \leq n)\ M1(x[j]) < M2(x[j]).$$

Definition: a model is said to be a *minimal model* iff it has no proper submodels.

Definition: the *problem lattice* is the product

```
L = L[1] x ... x L[n],
```

and on this lattice we have a monotone function F defined by

```
F(<x[1],...,x[n]>) = <x[1]\/e[1,1]\/e[1,2]\/...
                     ,x[2]\/e[2,1]\/...
                     ,...
                     ,x[n]\/e[n,1]\/e[n,2]\/...
                     >
```

where e[i,k] is the right hand side of the kth inequality with left hand side x[i]. Here is an example, taken from [Mellish 85]:

```
x[1] ≥ and(x[1],x[2])
x[2] ≥ 0
x[3] ≥ x[1]
```

so

```
F(<x,y,z>) = <x\/(x/\y),y\/0,z\/z>
           = <x,y,z/\x>
```

In this example, <0,0,0> is a fixed point of F. We want a method for finding such fixed points in general.

Now a lattice of finite depth is complete, and a product of finite lattices is complete, so L is complete. It is a standard result of lattice theory [Birkhoff 67] that the fixed points of a monotone function over a complete lattice form a complete lattice. Hence the fixed points of F over L form a complete lattice. In particular, there is a unique minimal fixed point, and this fixed point is a minimal model of the set of inequalities.

Definition: A *lattice equation problem* is a finite fixed-point problem where some or all of the inequalities are equalities, and the task is to find a minimal model or to show that there are no models.

It is easy to show that if a lattice equation problem has a solution it is the least fixed point of the corresponding finite fixed-point problem. So we can solve lattice equation problems (or determine that they have no solution) in linear time and space.

Definition: a monotone function f(x1,...,xn) is *top-preserving* if f(1,...,1) = 1.

If all the monotone functions in a lattice equation problem are top-preserving, then the interpretation which assigns the maximum element of its lattice to each variable is a model, and the problem is equivalent to the corresponding finite fixed point problem.

6.2. The Algorithm

To improve the performance of the algorithm, it is useful to reduce the inequalities to a normal form where every inequality has one of two forms:

1. x[i] ≥ k
   for some k in L[i]

2. x[i] ≥ f(x1,...,xp,k1,...,kq)
   where x1...xp are not necessarily distinct variables, k1...kq are not necessarily distinct constants, and f is one of the monotone functions (including the identity function).

This is done by evaluating constant expressions, and replacing each nested expression by a new variable and adding a new inequality. Normalisation can clearly be done in linear time and space. Inequalities of form (1) can be handled by initialising each variable to the least upper bound of all its constant lower bounds.

The algorithm uses the following data structures:

```
for i = 1..N
    value[i]  : the current value of variable x[i]
    chain[i]  : a list of all the simplified right-hand-
                sides in which x[i] appears
    link[i]   : points to the next element in the stack,
                is NIL for the bottom element, or is OUT
                when x[i] is not on the stack.

for j = 1..E
    lhs[j]    : index of a variable
    expr[j]   : x[lhs[j]] ≥ expr[j] is jth inequality
```

We use auxiliary variables

```
s         : top of stack pointer
i         : index of variable being processed
c         : points to (a tail of) chain[i]
j         : index of inequality being processed
t         : a lattice value
```

The algorithm has two phases, an initialisation phase, and a closure phase. The initialisation phase computes the initial value of each variable, and adds it to the stack if it is non-zero.

```
for i from 1 to N do
    value[i], link[i] := 0, OUT
od

s := NIL
for j from 1 to E do
    i := lhs[j]
    t := value[i] \/ eval(expr[j])
    if t > value[i] then
        value[i], link[i], s := t, s, i
    fi
od
```

The cost of the initialisation phase is proportional to the size of the problem; the loop on j dominates the loop on i.

After the initialisation phase is complete, all variables have assigned values, and the stack contains all the variables with non-zero values. Applied to the example from [Mellish 85], the initialisation phase terminates with all variables 0 and s NIL.

The closure phase keeps propagating the changed variables until no further changes occur.

```
while s ≠ NIL do
    i, s := s, link[s]
    link[i], c := OUT, chain[i]
    while c ≠ NIL do
        j, c := head[c], tail[c]
        i := lhs[j]
        t := value[i] \/ eval(expr[j])
        if t > value[i] then
            value[i] := t
            if link[i] = OUT then
                link[i], s := s, i
            fi
        fi
    od
od
```

The significance of finite depth is that each iteration of the outer loop corresponds to a *change* in an x[i]. So the depth of the whole problem bounds the number of iterations of the outer loop. The body of the inner loop takes constant time. So the next question is the number of iterations of the inner loop. Each chain is traversed in its entirety once for every time that the corresponding variable is stacked. Letting dmax be the maximum depth of any of the lattices, each variable can be stacked at most dmax times. So a bound on the number of iterations is

```
dmax x sum {i=1..N} length(chain[i])
```

But

```
sum {i=1..N} length(chain[i])
```

is just the number of variable occurrences in right hand sides, which is bounded by the size of the problem.

Accordingly, the total complexity of the algorithm is

```
O(d + E + N + dmax x size of problem)
```

But d $\leq$ dmax x N, and E + N < size of problem, so the cost is

```
O(dmax x size of problem)
```

For a fixed set of lattices, the computational cost is proportional to the size of the set of inequalities.

We have just proved

Theorem: any finite fixed point problem can be solved in linear time and linear space.

Theorem: any lattice equation problem can be solved in linear time and linear space.

6.3. Applications

This section describes some of the applications this algorithm is useful for, drawn from knowledge representation, logic programming, and graph theory.

6.3.1. Functional Dependencies

The original LINCLOSURE algorithm was developed to handle *functional dependencies* in relational data bases. See [Maier 83] chapter 4 for a good explanation. Given a relation such as

```
assign(Pilot, Flight, Date, Departs)
```

a functional dependency such as

```
{Flight, Date} -> Pilot
```

means that

```
for all Flight, Date
   there is at most one Pilot such that
      there is a Departs such that
         assign(Pilot, Flight, Date, Departs)
```

That is, if we just look at the Pilot, Flight, and Date arguments, 'assign' is a partial function from Flight and Date to Pilot.

It is possible to state functional dependencies in ABASE. This feature was copied from MECHO, though the syntax is different. One would write

```
unique(assign(P,F,D,T), [F,D], [P]).
```

But despite the different syntax, it is just an ordinary functional dependency. Given a set of functional dependencies, it is useful to find *all* the functional dependencies which they logically entail. The LINCLOSURE algorithm is useful for doing this efficiently.

6.3.2. Belief Revision

Whenever ABASE completes an inference, it records a justification. The use of propositional Horn clauses for belief revision was described at the beginning of this chapter.

6.3.3. Logic Programs With Uncertainties

If, despite the fundamental problems with attaching uncertainties to predications described in chapter 4, you decide to use that approach anyway, the methods of [Shapiro 83] can usefully be generalised to complete lattices. (So can "Fuzzy Set Theory". See [Erceg 76].)

Any *instance* of an "uncertain Horn clause"

```
f: C <- H1 & ... & Hn
```

corresponds to an inequality relating the uncertainties associated with the hypotheses and conclusion:

```
cf(C') ≥ f(cf(H1'),...,cf(Hn'))
```

If the certainty space we are using has finite depth, we can propagate changes in the certainty factors of facts and rules in time linear in the size of the proof so far. Or, if we are interested only in the certainty of one conclusion, linear in the size of the proof of that conclusion. Note that [Shapiro 83] uses the interval (0,1] as certainty space, which is *not* of finite depth.

If all the rules in a logic program with uncertainties are in fact propositional, then the cost of propagating uncertainties is at worst linear in the size of the rule base.

It is possible to extend the logic programs with uncertainties approach to include "negated" hypotheses, where a negated hypothesis has the form

```
cf(H) ≤ k
```

where k is a lattice element. Such a subgoal is either absolutely true, absolutely false, or undetermined, and is not itself uncertain. This is analogous to the use of finite failure as a substitute for negation in ordinary logic programming, and is subject to similar limitations. In particular, the addition of new facts or the increase of a certainty may *falsify* a negated hypothesis, so the modified LINCLOSURE does not apply directly to logic programs with both uncertainties and negation.

6.3.4. Description-to-Description Rules

A conditional description to description rule (CDD rule) has the logical form

```
if rule_enabled(X) and
   description(X) ≤ D and
   D ≤ d0 and
   D /\ d1 = 0 & ... & D /\ dn = 0
then
   description(X) ≤ f(D)
```

where f is a monotone function. For a particular X at a particular time, the set of enabled CDD rules forms a finite fixed point problem.

If, as is usually the case, the description space we are using is a product of several aspects, we can treat each of these aspects as a separate variable in the finite fixed point problem. (That is in fact how ABASE stores aspects.) An interesting result of this section is that the cost of completing a description depends on the depth of the deepest aspect lattice, rather than on the depth of the product lattice as a whole.

More importantly, the form of CDD rules in ABASE was restricted to what I thought could be implemented efficiently. But the fact that finite fixed point problems can be solved in linear time means that I was unduly pessimistic. Consider the following generalised schema (for Generalised DD rules):

```
if rule_enabled(X1,...,Xn) and
   description(X1) ≤ D1 and D1 ≤ d1 and ...
   ... and
   description(Xn) ≤ Dn and Dn ≤ dn and ...
then
   description(Xi) ≤ f(D1,...,Dn)
```

An example of such a rule is

```
   mother(Parent, Child) &
   the variety of Parent is S
-> the variety of Child is weaken_to_species(S).
```

where weaken_to_species is a function which maps varieties (such as Dachshund or Samoyed or Siamese or Manx) to species (such as *canis familiaris* or *felis domesticus*), and leaves classifications at or above the species level alone.

We can handle rules like this by constructing a finite fixed-point problem *dynamically*. Whenever we learn about a new individual, we create a variable for each of its aspects. Whenever an enabling condition is proven, we add the corresponding instance of the GDD rule as an inequality. For example, when we first hear of Samuel, we might create the variable

```
variety_of_samuel
```

among others. When it is proven that mother(hannah,samuel), we add the inequality

```
variety_of_samuel ≤
  weaken_to_species(variety_of_hannah)
```

If we update many descriptions at once, the cost of propagating the consequences of these through the instantiated and enabled GDD rules is proportional to the total bulk of the rules. This really pays off when combined with belief revision: we can economically revise the descriptions by setting all the variables to their initial values and re-running the finite fixed point algorithm.

6.3.5. Transitive Closure

Suppose we are given an NxN matrix **M** whose elements come from a lattice of finite depth, and we want to compute the generalised transitive closure

$$\lim_{k\to\infty} M_k$$

where $M_0 = M$

$M_{k+1} = M_k \;\backslash/\; M_k * M_k$

where $(X*Y)_{ij} = \backslash/_{p=1..N} f(X_{ip}, Y_{pj})$

f being any monotone increasing function. For the ordinary transitive closure, the lattice is {0,1}, and f is $\wedge$.

Letting **C** be the desired solution matrix, we can convert this to a finite fixed-point problem as follows:

```
for i = 1..N, j = 1..N
    C_ij >= M_ij
for i = 1..N, J = 1..N, p = 1..N
    C_ij >= f(C_ip, C_pj)
```

Clearly, the size of this finite fixed-point problem is proportional to N^3, regardless of the value of **M**. Whatever the function f, we can solve this problem in cubic time.

For the ordinary transitive closure, this algorithm can be coded in C [Harbison & Steele 84, Kernighan 78] as follows:

```
struct {unsigned char v, r, c;}
    C[NMAX+1][NMAX+1];

void init(n)
    register int n;
    {
        register int i, j;

        for (i = n; i > 0; i--)
            for (j = n; j > 0; j--)
                C[i][j].v = 0;
    }

void arc(i, j, n)
    register int i, j;
    int n;
    {
        register int k;
        register int si, sj;

        if (!C[i][j].v) {
            C[i][j].v = 1,
            C[i][j].r = 0, C[i][j].c = 0;
            for (; i != 0; i = si, j = sj) {
                si = C[i][j].r, sj = C[i][j].c;
                for (k = n; k > 0; k--) {
                    if (C[j][k].v && !C[i][k].v)
                        C[i][k].v = 1,
                        C[i][k].r = si, C[i][k].c = sj,
                        si = i, sj = k;
                    if (C[k][i].v && !C[k][j].v)
                        C[k][j].v = 1,
                        C[k][j].r = si, C[k][j].c = sj,
                        si = k, sj = j;
                }
            }
        }
    }
```

Algorithms for computing the transitive closure in cubic time are already known, most notably Warshall's algorithm [Warshall 62]:

```
for j from 1 to N do
    for i from 1 to N do
        C[i,j] := M[i,j]

for j from 1 to N do
    for i from 1 to N do
        if C[i,j] then
            for p from 1 to N do
                if C[j,p] then
                    C[i,p] := true
```

Warshall's algorithm can be implemented very efficiently indeed. But this new method has an advantage: it is *incremental*. That is, suppose we start with a particular M and compute C. Now, we can *add* new arcs to M (by setting some M[i,j] to a higher value than it previously had), and update C by resuming the closure phase of my algorithm, and the total time for computing the closure is *still* cubic in N.

Transitive closure is actually a useful operation for expert systems which can have recursive rules. In ASA, for example, "is contained in" is a transitive relation between experiment stages. There are a number of base relations (such as one step being the first sub-step of a sequence step) which act as M, and the derived relation acts as C. Suppose that we have a relation

```
c(X,Y) <- m(X,Y)
c(X,Y) <- c(X,Z) & c(Z,Y)
```

There are several ways we could handle this. One way is to use a general deduction method. ABASE uses Earley deduction [Pereira & Warren 83] which can easily handle such rules. If there are N entities of the appropriate type, this can cost $O(N^2)$ per query of c. An improvement on this approach is to cache the results of queries of c. Since there are at most N^2 distinct queries, this yields a total cost of $O(N^4)$. The implementation of Earley deduction in ABASE caches all results, so computing a transitive closure this way in ABASE costs $O(N^4)$ inference steps. Separating transitive relations out and using this algorithm on them yields an incremental method (of importance because the base relation is incrementally acquired) with total cost $O(N^3)$, and the constant factor is smaller too.

ABASE does not use this algorithm, for the simple reason that I didn't invent it in time. For the small problems that ASA tackled, the use of Earley deduction with caching was satisfactory. But almost any technique will work on small problems, and having a fast algorithm for transitive closure is important if we are to cope with large problems.

Even if ABASE used this algorithm, it would not be the best way to handle the "is contained in" relation in ASA. The reason for that is that experiment steps in ASA's representation cannot overlap, so "is contained in" is a *tree*. Accordingly, the fast tree algorithm

presented for type trees in chapter 7 can be adapted to yield an O(N) algorithm for this case, which is much better than $O(N^3)$. For acyclic graphs in general, we can compute transitive closures in $O(N^2)$ time.

6.3.6. Optimising Logic Programs

Finally, Chris Mellish has shown in [Mellish 85] that several interesting properties of Prolog predicates can be derived by forming a finite fixed point problem (whose size is roughly proportional to the size of the original program) and solving it.

After preprocessing, all of the rules that ABASE can handle are either special annotations (handled at "compile time" by special purpose closure methods such as LINCLOSURE), or generalised DD rules, or else Horn clauses. Methods for analysing sets of Horn clauses are therefore relevant to analysing Knowledge Bases written in first order logic.

6.4. Function-Free Horn Clauses

The family of algorithms I have described so far in this chapter is derived from the LINCLOSURE for solving sets of propositional Horn clauses. It would be pleasant indeed if we could use this basic idea in a more general setting.

The basic knowledge representation language used by ABASE is function-free Horn clauses. The inference method used by ABASE is *Earley Deduction* [Pereira & Warren 83]. Earley Deduction is a family of proof procedures for sets of Horn clauses; it includes Prolog as an extreme member. The variant used by ABASE can be regarded as a blend of backward chaining, deciding which goals are "interesting", and forward chaining, deciding which "interesting" facts are true. There are in fact several top level goals: one is to prove that the client can be satisfied, and the others are checking for inconsistencies. Earley deduction was chosen as the inference method so that this blend of backward chaining and forward chaining could be used: as soon as a fact was asserted by the client (whether the backward chaining procedure was ready for it or not) it will be forward chained from, so that "obvious" inconsistencies will be spotted promptly. Also, the variant of Earley Deduction used can easily accommodate non-chronological backtracking. Earley Deduction was not chosen to make inference easier, but because it provides a much better basis for managing a "conversation" than pure backward chaining.

It is possible to regard the ABASE inference engine as consisting of three levels:

- The backward chaining level works backwards from the top level goals, deciding which goals are interesting, and creating "dotted rules". This level decides when to ask questions, when to switch off goals because they are no longer inter-

esting, what question to ask next, when an assumption can be made, and when to create a new entity.

- The forward chaining level works forwards from the facts asserted by the client or guessed by the default mechanism, chaining not with the original rules, but with the "dotted rules" created by the backward chaining level.
- The dependency maintenance level uses the propositional clauses (dependencies) created by the forward chaining level to keep track of which facts are still supported by the client's current assertions.

We have already seen how to handle the propositional Horn clauses belonging to the third level efficiently. Is there an efficient way to handle the function-free Horn clauses belonging to the second level?

Suppose we have a set of function-free Horn clauses in which K distinct constants appear. Let the maximum number of distinct variables in any clause be V, and let the size of the set by N symbols. Let the query

```
<- G1 & ... & Gn
```

be included as the only rule defining 'ANSWER':

```
ANSWER(vars...) <- G1 & ... & Gn
```

and let this clause be included in the K, V, N figures. Then there are at most K^V different ground instances of any rule, so the entire set of clauses is equivalent to set of propositional Horn clauses (treating each different ground instance of a logical atom as a different propositional letter) of size at worst (K^V)*N. We can solve that set in time O((K^V)*N).

That is a brute force approach. Unfortunately, it is not always possible to do better. For example,

```
p(c11, c12) <-
...
p(cn1, cn2) <-

q(X, Y) <- p(X, Y)
q(X, Y) <- q(X, Z) & q(Z, Y)
```

This is precisely the transitive closure problem we studied earlier, and we saw that it takes O(K^3) time where K is the number of distinct cij, and here V=3.[10]

One reason why this example takes a lot of work is that it has a lot of solutions. What we are looking for is a forward chaining method which takes time proportional to the number of solutions found, or not too much worse than that.

---

[10]The transitive closure problem can be solved in O(K^2.5ish) time, so this doesn't *prove* that we can't do better than O((K^V)*N).

Now a predicate symbol of arity N can be regarded as a variable ranging over the lattice 2^(K^N), K being the set of constants. This is a lattice of finite (but very large) depth. A function-free Horn clause

```
p(x1,...,xn) <-
    q1(y11,...,y1k1) & ... &
    qm(ym1,...,ymkm)
```

can be regarded as an inequality

```
p ≥ f(q1,...,qm)
```

for a suitable monotone function f. If the cost of evaluating f(q1,...,qm) were proportional to the size of the rule in question, which it is **not**, we would then have a simple finite fixed-point problem. Each rule will be fired when one of its qi has been changed, and this can only happen when at least one new solutions has been found for that qi, and the cost would then be O(size of rule set x number of solutions).

The trouble is the fact that evaluating the right hand side of a rule can be very costly indeed. The point of OPS5 [--- 82], for example, is that it has an efficient method for doing this.

A normal form for the set of clauses has each clause in one of three forms:

```
0.  H <-
1.  H <- G1
2.  H <- G1 & G2
```

This can be done without increasing the number of conjunctions in the clauses, and with only a linear increase in the number of predicate letters. For example,

```
says_knave(P1, P2, YN1) <-
        believes_knave(P1, P2, YN2) &
        is_knave(P1, YN3) &
        combine(YN2, YN3, YN1).

believes_knave(P1, P1, YN1) <-
        is_knave(P1, YN1)

believes_knave(P1, P2, YN1) <-
        says_knave(P3, P2, YN2) &
        believes_knave(P1, P3, YN3) &
        combine(YN2, YN3, YN1)
```

might be normalised as

```
says_knave(P1, P2, YN1) <-
        believes_knave(P1, P2, YN2) &
        says__1(P1, YN2, YN1)

says__1(P1, YN2, YN1) <-
        is_knave(P1, YN3) &
        combine(YN2, YN3, YN1)

believes_knave(P1, P1, YN1) <-
        is_knave(P1, YN1)
```

```
believes_knave(P1, P2, YN1) <-
        says_knave(P3, P2, YN2) &
        believes__1(P1, P3,  YN2, YN1)

believes__1(P1, P3, YN2, YN1) <-
        believes_knave(P1, P3, YN3) &
        combine(YN2, YN3, YN1)
```

There are several other possible normalisations of these three clauses.

Determining the best normalisation of a set of clauses requires additional information about the expected sizes of projections of the predicates. Given that information, it is a generalisation of the well known "matrix chain product" problem (see [Sedgwick 83], pp 486-489). That is, deciding whether to normalise

```
p(A, D) <- q1(B, A) & q3(C, D) & q2(B, C)
```

as

```
p(A, D) <- p__1(A, C) & q3(C, D)
p__1(A, C) <- q1(B, A) & q2(B, C)
```

or as

```
p(A, D) <- p__2(B, D) & q1(A, B)
p__2(B, D) <- q3(C, D) & q2(B, C)
```

is very like deciding whether to calculate

$$X^i_j = U^i_m V^m_n W^n_j$$

as

$$X^i_j = (U^i_m V^m_n) W^n_j$$

or as

$$X^i_j = U^i_m (V^m_n W^n_j)$$

This is known to be a very hard problem (there are $O(4^N.N^{-3/2})$ bracketings to try), but fortunately we only have to do it once, and typically there aren't many goals in a rule. The dynamic programming algorithm given in [Sedgwick 83] takes $O(N^3)$ time. Exploiting functional dependencies is very important here!

6.4.1. Unit Clauses

Rules of the form

```
0.  H <-
```

are taken care of by the initialisation phase.

6.4.2. Doublet Clauses

Rules of the form

```
1.  H <- G1
```

introduce the first difficulty: finding them. Having proved or been given a new function-free fact G, how do we find which rules of this form it is relevant to (which G1 unify with G)? When we considered propositional Horn clauses, all the clauses containing the same predicate letter were relevant, but this is no longer the case. (At least, it is no longer the case for the original predicate symbols. The predicate symbols introduced by normalisation are relevant to all the rules they appear in, as the atoms in which they appear have only variables as arguments.)

The problem is this: we are given a fixed set S of function-free atoms -- the atoms which appear in rule bodies. We have another function-free atom A with the same predicate symbol -- the fact which has just been proven or volunteered by the client. We want to identify the members of S which unify with A, and we want to do this rapidly.

The first thing to note is that the predicate symbol is known for each of the atoms in S. So in unit time (using a hash table) we can locate the atoms in S with the same predicate letter as A. Now we are considering

```
A = p(A1,...,An)
Sp = {..., p(S1,...,Sn), ...}
```

We can determine whether p(A1,...,An) and p(S1,...,Sn) unify in o(n) time. So we can find all the relevant members of S in time at worst o(n.|Sp|). If R is the subset of Sp containing all the atoms which unify with A, we have to inspect every member of R, so the best we can do is o(n.|R|). A good method for searching the table is that in [Ramamohanorao 86]. Other indexing methods are available. ABASE doesn't try to be clever for the simple reason that virtually all the goals in ASA's rules have all the arguments unbound, so that Sp is typically a singleton set.

6.4.3. Triplet Clauses

Rules of the form

```
2.  H <- G1 & G2
```

have also to be found. For simplicity, suppose we store each such rule twice, as

```
2a. H <- G1 & G2
2b. H <- G2 & G1
```

We now have rules of the form

```
Conclusion <- Cue & Context
```

Given a new fact, we want to find rules with a unifying Cue. We can do this using

[Ramamohanorao 86] or any other such scheme. We now have an instance of Context, say Context', and for every fact matching Context' we want to derive the corresponding instance of Conclusion.

This searching for a Context in the data base of facts is logically the same as the problem of finding the relevant members of S given A. However, the data base is typically large and growing. Fortunately, [Ramamohanorao 86] is a dynamic method, so can still be used.

A cruder method was devised for ABASE but not installed. The data base was to be broken into two subsets: the ground facts and the non-ground facts. (In practice there are no non-ground facts, but there is nothing in the problem specification to exclude them.) Non-ground facts were to be handled by the brute-force method. Ground facts were to be handled by having 2^n hash tables for each predicate symbol of arity n. Each index $0..2^n-1$ specifies a subset of the arguments: for each fact and index the appropriate subset of the arguments of the fact were to be used as the key for inserting the fact in that table. When an instance of Context was to be looked up, the hash table corresponding to the bound arguments would be used. [Ramamohanorao 86] has superceded this method.

6.4.4. Overall Performance

Roughly speaking, the work which this method performs is (with optimal indexing) at least proportional to the sum of sizes of all true instances of rules. For example, in the transitive closure problem

```
ancestor(ci, cj) <-
...
ancestor(X, Y) <- ancestor(X, Z) & ancestor(Z, Y)
```

the sum of the sizes of all true rule instances can be cubic in the number of facts.

6.4.5. Relevance to ABASE

ABASE doesn't quite do this. As backwards chaining proceeds, "dotted" rules are produced. A dotted rule has the form

```
(Conclusion <- {Done} & Cue & Context)theta
```

where Done is a conjunction of the goals which have already been solved, theta is the substitution which resulted from their solution, Cue is the goal which has been selected, and Context is the remaining goals. (Cue)theta is set up as a subgoal. A dotted rule with empty Context (or rather, whose context consists entirely of built in predicates, type tests, and Prolog escapes) corresponds to a doublet clause. A dotted rule with nonempty Context corresponds to a triplet clause. Thus the forward-chaining rule set grows as backward-

chaining proceeds. If the knowledge base contains N negative rules, the state of the Earley Deduction engine is a merge of the parallel execution of N+1 processes, one for each negative rule and one for the consultation.

The benefit of working from the dotted rules rather than the original rules is that backward chaining focusses forward chaining: only "interesting" conclusions are drawn. The price is that we don't get the opportunity to optimise the bracketing of general clauses into triplet form.

6.5. Summary

In this chapter I have presented an algorithm which finds the solution of a finite fixed-point problem in linear time and space. The algorithm has many applications in logic-based expert systems. It is related to the LINCLOSURE algorithm, but appears to be new. From it one can derive an incremental transitive closure algorithm which also appears to be new.

If we let the size of a problem be S, and the number of variables be V, it is straightforward to program the algorithm in Prolog so as to take O(S) space and O(S.lgV) time. This is optimal in a pointer machine model.

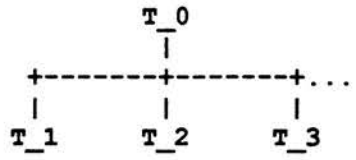
The LINCLOSURE idea was applied to forward chaining on function-free Horn clauses. Given an efficient method of locating function-free atoms in a dynamic data base which unify with a given atom, this yields a method whose cost is proportional to the sum of the sizes of all true instances of (normalised) rules.

Chapter 7
A New Data-Structure for Type Trees

7.1. Introduction.

Type trees are easy to understand, they encode a lot of information in a compact way, and type information can be used both to check that rules and new facts make sense and to reduce the size of the search space. Making them even more efficient is therefore a Good Thing.

A taxonomy is an arrangement of classes (types, unary predicates) into a tree. A node in the tree and its outgoing arcs, such as



represents a set of axioms

```
(∀x) T_1(x) ⇒ T_0(x)
(∀x) T_2(x) ⇒ T_0(x)
(∀x) T_3(x) ⇒ T_0(x)
...
(∀x) T_1(x) ⇒ ~T_2(x) & ~T_3(x) & ~...
(∀x) T_2(x) ⇒ ~T_3(x) & ~...
...
```

For convenience we may call these XOR nodes, though the relation is not strictly an exclusive or.

We typically want to use such a set of axioms to do propositional reasoning about the type of a known individual. So we want to ask questions like

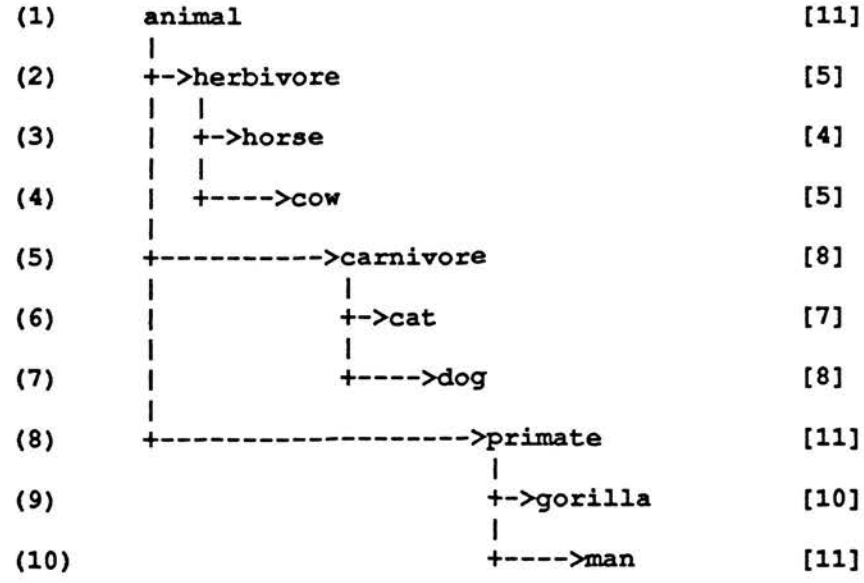
```
Given T_1(x), is T_2(x)
     (a) certain?
         That is, is T_1 a sub-type of T_2?
     (b) possible?
         That is, is T_1 a super-type of T_2?
     (c) impossible?
         That is, are T_1 and T_2 incompatible?
     (neither of the above)
Given T_1(x),
     (d) which T(x) follow?
         That is, find the super-types of T_1.
     (e) which T(x) are still possible?
         That is, find the sub-types of T_1.
Given T_1 and T_2,
     (f) what is their nearest common supertype?
```

The data structure presented in this chapter implements these operations in asymptotically optimal time and space. The only operation which is slowed down is adding a new type to the tree, and that remains constant *expected* time.

7.2. Numbering the nodes.

The key idea is that we can number the nodes of the type tree so that the numbers tell us a good deal about the structure. Here is an example.

ord tree ord(suc)



The successor of the root is a dummy node. The successor of any other node is its right brother if it has one, otherwise its father's successor. Successor links are related to threaded trees [Knuth 73] but are not the same. The node numbers are just the numbers assigned by a depth-first pre-order traversal. How do these numbers help?

If T1 and T2 are nodes in the tree representing types, then

(*) **T1 is a subtype of T2 iff**
ord(T2) ≤ ord(T1) < ord(suc(T2)).

That's all there is to the idea. This directly answers questions (a), (b), and (c). Question (e) can be answered by linking a node to its sons, but another way is to link each node to the node with the next higher number, and to use (*) to stop at the first non-descendant. (See the function appBelow below.) Question (d) is easily answered by linking each node to its father (see the function appAbove). Father links also let us answer question (f) cheaply. To find the common super-type of types T_1 and T_2, we just scan the ancestors of one node to find the first one which is an ancestor of the other:

```
while not(subtype(T_2,T_1)) do T_1 := father(T_1);
```

If T_1, T_2 are at depths D1, D2 in the tree, and the common super-type is at depth D, the cost of finding the super-type is thus proportional to D1-D. The function comtype below saves one of the comparisons in each subtype test but has cost proportional to max(D1,D2)-D. We could store the depth of each node and make the cost proportional to min(D1,D2)-D.

7.3. The Complete Method.

The scheme presented in the previous section requires a fixed set of nodes. It can be useful for an Expert System to acquire new taxonomic information from its client during a consultation, but adding a node to the tree as presented so far means renumbering most of the nodes. The question is whether we can relax the numbering system to make it easier to update, without losing its other properties. The answer is that we can. All we need from the numbers is their *order*, so we can leave gaps between the numbers of successive nodes. New nodes can then go in the gaps. Only rarely do we need to renumber a subtree.

I have chosen to present the complete method in the form of working code in the language C. Providing these algorithms to the community is the major point of this chapter. Those unfamiliar with C should consult [Kernighan 78, Harbison & Steele 84]. Failing that, here is a summary. C is a Pascal-like language. "struct" declares a record type. "int" means "integer", and is also used for boolean (0 = false, anything else = true). "{" and "}" are used for "begin" and "end". "test ? if_true : if_false" is a conditional expression, whose value is if_true if test is non-zero(true) or if_false if test is 0(false). The "for" statement is a disguised "while":

```
        for (Init; Test; Step) Body
means
        { Init; while (Test) { Body; Step; } }
```

The most idiosyncratic aspect of C is the way it handles pointers. If V is a variable of type T, then &V is a pointer value of type *T, pointing to the variable. If P is a pointer variable of type *T, then *P is the variable that P points to. If P is a pointer to a record(struct) with a com-

ponent f, P->f is the f component of the record P points to (in Pascal, P^.f). 0 is the null pointer value. With this sketchy background, let's get on with the program.

```
typedef struct node *nodep;
#define Nil (nodep)0

struct node
    {
        nodep Nxt;      /* node with next higher number */
        nodep Dad;      /* the father of this node */
        nodep Yds;      /* this node's Youngest DeScendant*/
        int   Ord;      /* the magic ordering number */
        int   Lab;      /* the label, for external use */
    };
```

A node is represented by a record. All the nodes in a tree are linked together by their Nxt fields, in the same order as that given by the Ord numbers. The Ord numbers do not necessarily go up in steps of 1; but their relative order is the same depth-first pre-order traversal order. Having gaps in this order is what lets us insert new nodes without updating all the numbers.

The youngest descendant of a node is the descendant with the highest number, and is the most recently added one. As such, it must be a leaf. It turns out that we do not need to store a successor link, as the successor of a node (oldest younger brother if there is one, otherwise father's successor) is always N->Yds->Nxt. When we add a new son to a node N, it will be inserted between N->Yds and N->Yds->Nxt, and will become N's youngest descendant.

Here is how the animals tree might be represented.

Node	Dad	Yds	Nxt	Ord
animal	-	man	herbivore	1
herbivore	animal	cow	horse	2
horse	herbivore	horse	cow	3
cow	herbivore	cow	carnivore	501
carnivore	animal	dog	cat	750
cat	carnivore	cat	dog	751
dog	carnivore	dog	primate	875
primate	animal	man	gorilla	937
gorilla	primate	gorilla	man	938
man	primate	man	DUMMY	969
DUMMY	-	-	-	1000

The same dummy node can be used for all trees, so it is predeclared.

```
struct node dummy =
    {
        Nil,              /* Nxt, never inspected */
        Nil,              /* Dad, never inspected */
        Nil,              /* Yds, never inspected */
        1000000000,       /* Ord, upper bound for each tree */
        0                 /* Lab, never inspected */
    };
```

To create a new tree whose root is labelled Lab, we say

```
root = mkRoot(Lab);
```

where

```
nodep mkRoot(lab)
    int lab;
    {
        nodep It = (nodep)malloc(sizeof (struct node));
        It->Nxt = &dummy,
        It->Dad = Nil,
        It->Yds = It,
        It->Ord = 0,      /* lower bound for whole tree */
        It->Lab = lab;
        return It;
    }
```

The links enable us to find all the ancestors and all the descendants of a node. Although we do not store son links, the fact that the Youngest Descendant links let us find younger brothers means that we can actually find all the sons of a node quite easily. It turns out that

```
N is a leaf iff N->Yds = N
if N is not a leaf, N->Nxt is N's oldest son.
```

So we have three routines:

```
void appAbove(N, P)       /* apply P to all ancestors of N */
    nodep N;              /* including N itself */
    void (*P)();          /* P is a procedure */
    {
        while (N != Nil) (*P)(N), N = N->Dad;
    }

void appSons(N, P)        /* apply P to all sons of N */
    nodep N;
    void (*P)();
    {
        nodep S = N->Yds->Nxt;  /* successor of N */

        N = nxt(N);             /* oldest son */
        while (N != S) (*P)(N), N = N->Yds->Nxt;
    }

void appBelow(N, P)       /* apply P to all N's descendants */
    nodep N;              /* including N itself */
    void (*P)();
    {
        nodep S = N->Yds->Nxt;
```

```
        while (N != S) (*P)(N), N = N->Nxt;
    }
```

These routines traverse the required sections of the tree with constant extra storage (at most one pointer), and at cost a+b|Examined|, where Examined is the set of nodes inspected, and a,b are small constants. Their correctness is less obvious, and depends largely on the definition of Youngest Descendant and successor.

Exhibiting these routines merely shows that this data structure is no worse than a threaded tree (which also requires either 3 pointers or 2 pointers and a "leaf?" flag per node). The real point of this data structure is to make type comparison fast. The next three routines do this. They all assume that T1 and T2 belong to the same tree.

```
int subtype(T1, T2)
    /* is T1 a subtype of T2 (includes T1=T2)? */
    nodep T1, T2;
    {
        return T2->Ord <= T1->Ord
            && T1->Ord < T2->Yds->Nxt->Ord;
    }

/*  reltype(T1, t2) returns
    '=' if T1 and T2 are the same;
    '<' if T1 is a proper subtype of T2;
    '>' if T1 is a proper supertype of T2;
    '#' if T1 and T2 are incompatible.
*/
char reltype(T1, T2)
    /* works out what relation holds between T1 and T2 */
    nodep T1, T2;
    {
        return
           T1->Ord < T2->Ord ?
                (T2->Ord < T1->Yds->Nxt->Ord ? '>' : '#')
         : T1->Ord > T2->Ord ?
                (T1->Ord < T2->Yds->Nxt->Ord ? '<' : '#')
         : '=';
    }

nodep comtype(T1, T2)
    /* returns the smallest common supertype of T1 & T2 */
    nodep T1, T2;
    {
        int O2 = T2->Ord;

        if (T1->Ord > O2) O2 = T1->Ord, T1 = T2;
        while (T1->Yds->Nxt->Ord <= O2) T1 = T1->Dad;
    }
```

This is all very pretty, but if it is to be any use, building the tree has to be affordable. Suppose we want to add a new node S as a son of node D. The first thing to do is to maintain the links. The point of the Youngest Descendant link is that it points to the node after which the new node is to be inserted. So doing

```
S->Nxt = D->Yds->Nxt, D->Yds->Nxt = S;
```

is sufficient to insert S into the Nxt chain. But we also have to maintain the Youngest Descendant links themselves. We thus arrive at

```
nodep addSon(D, lab)     /* links only version */
    nodep D;
    int lab;
    {
        nodep S = (nodep)malloc(sizeof (struct node));
        nodep Y = D->Yds;

        S->Nxt = Y->Nxt,
        S->Dad = D,
        S->Yds = S,
        S->Lab = lab;
        while (D != Nil && D->Yds == Y)
            D->Yds = S, D = D->Dad;
        Y->Nxt = S;
        return S;
    }
```

The body of the while loop must be executed at least once. Thereafter it will be executed as long as D is its father's youngest son. If the tree has branching factor B > 1, the average number of times the loop is executed will thus be 1 + 1/B + 1/B^2 + ... = B/(B-1). Thus if B = 2, which is the smallest branching factor that makes sense for a type tree, the loop will be executed an average of twice. The worst case is when every node is the rightmost node in the tree at the time it is added, when all its ancestors will be visited. The worst way to build a tree is thus depth-first, when the cost per node is o(lgN) for a tree of N nodes, and the best way is breadth-first, when the cost is O(1) per node.

If Y->Nxt->Ord > 1+Y->Ord in the algorithm above, there is no difficulty in assigning a new number. ((Y->Nxt->Ord)+(Y->Ord))/2 will do fine. When S is D's first son, Y=D, and Y->Ord+1 will do even better. The animals tree showed this numbering at work. It has an unpleasant tendency to bunch the numbers up at the right. Renumbering removes this bias, and so reduces the likelihood of future renumbering.

If Y->Nxt->Ord = 1+Y->Ord, we have to renumber part of the tree. The general scheme is

```
while (D is infeasible) D = D->Dad;
renumber the tree rooted at D;
```

Let's look at the latter operation first, as it is easier, and it will tell us when a node is feasible.

To renumber the tree rooted at a particular node, we simply walk down the Nxt list as we do in appsubs, storing a number and incrementing it. We could use the same increment for all the nodes, but we can do slightly better. Because we always insert after D->Yds, and D->Yds is always a leaf, we will never have to insert anything immediately after an internal

node. So after internal nodes we can increment by 1, while after leaves (which can have new things inserted after them) we want to increment by as big a number as possible. So we have

```
void renumber(D, I)
    nodep D;            /* top of subtree to renumber */
    int I;              /* increment for leaf nodes */
    {
        nodep S = D->Yds->Nxt;  /* as in appBelow */
        int O = D->Ord;         /* D->Ord will not change */
        while (D != S) {
            D->Ord = O;
            O = O + (D->Yds == D ? I : 1);
            D = D->Nxt;
        }
    }
```

The condition for feasibility is thus that there should be an integer I > 0 for which D->Yds->Ord will end up less than D->Yds->Nxt->Ord. The easy way to test this is to determine the largest possible value of I and see if it is positive. So we get

```
int feasible(D, I)
    nodep D;            /* top of subtree to be tested */
    int *I;             /* the increment returned here */
    {
        nodep S = D->Yds->Nxt;
        int G = S->Ord - D->Ord;
        int L = 0;      /* number of leaves */

        while (D != S) {
            if (D->Yds == D) L = L+1; else G = G-1;
            D = D->Nxt;
        }
        /* G is the amount to distribute among leaves */
        return (*I = G/L) > 0;
    }
```

Since the new node is a descendant of D, and is a leaf, L must be non-zero when the division is done.

The final version of addSon is thus

```
nodep addSon(D, lab)
    nodep D;
    int lab;
    {
        nodep S = (nodep)malloc(sizeof (struct node));
        nodep Y = D->Yds;
        nodep A;
```

```
    S->Nxt = Y->Nxt,
    S->Dad = D,
    S->Yds = S,
    S->Lab = lab;
    for (A = D; A != Nil && A->Yds == Y;
                A->Yds = S, A = A->Dad) ;
    Y->Nxt = S;
    if (Y->Nxt->Ord - Y->Ord == 1) {
        int I;
        for (A = D; !feasible(A, &I); A = A->Dad) ;
        renumber(A, I);
    } else {
        S->Ord = Y == D ? 1+Y->Ord
                        : (Y->Nxt->Ord + Y->Ord)/2;
    }
    return S;
}
```

If we have to go up several generations before finding a feasible ancestor, the work testing the intermediate ancestors will have been wasted. For large branching factors this is negligible, while for small branching factors most of the nodes are only children and little renumbering is needed anyway.

The analysis of addSon is beyond my skill, so I generated lots of random trees ranging from 100 to 70,000 nodes, with a variety of branching factors. It proved extraordinarily difficult to provoke renumbering. Lest this seem surprising, recall that the total range available to the Ord numbers was 1000,000,000, which is very much bigger than 70,000. Without renumbering, a tightly coded version of addSon took about 0.2 ms per node on a VAX-11/750, exclusive of malloc() time. Even when renumbering occurred, the time didn't change much. Obviously, this method only works well when dummy.Ord is much bigger than N, but it is certainly practical for 100,000 nodes.

7.4. Multiple Context Data Bases

A *multiple context data base* is essentially a function

```
db : Key x Context -> Value
```

We are not concerned with the nature of Keys here. It is assumed that a fast data structure mapping keys to values exists, so that we can implement a multiple context data base as

```
Key -> (Context -> Value)
```

i.e. if we want to know the value associated with key K in context C we can use the key data structure to locate a context data structure:

```
db(K,C) = context_lookup(key_lookup(K),C).
```

The idea behind multiple context data bases is that keys and contexts are very different things: keys are associated with many fewer values than there are contexts, so it is worthwhile trying to store the data base in a compressed form.

Multiple context data bases differ according to the structure of the context space. Here I consider only tree-structured context spaces, where the operations are

```
root : Context
        returns the root of the context tree

addSon : Context -> Context
        adds a new subcontext to the given context,
        and returns the new context.

change : Key x Context x Value ->
        changes the value associated with a key in a
        specific context.

lookup : Key x Context -> Value
        returns the value associated with a key
```

Notionally, we maintain a set of (Context,Value) pairs for each key. So

```
change(K, C, V)
    pairs[K] := {(c,v) in pairs[K] | c ≠ C}
                ∪ {(C,V)}

lookup(K, C)
    let (c,v) be the pair in
        {(c1,v1) in pairs[K] | c1 is an ancestor of C}
        with deepest c component
    return v
```

We could actually implement these operations in exactly this way. The trouble with that is that the cost of a change or lookup is proportional to the number of contexts in which the key has been assigned a value. A heuristic which has been used in some implementations is to number the nodes of the context tree as they are created; this is a topological ordering of the tree. The set of changes is then kept in decreasing order of context. That may allow the use of binary search in "change", but "lookup" is still sequential.

With the node numbering scheme presented in this chapter, a sub-tree corresponds to an interval of node numbers. Indeed, to each node there corresponds an interval [l,u). So we could regard the set of (Context,Value) pairs as a set of (Interval,Value) pairs. In this view,

```
lookup(K,C)
    let n be the node number of C
    let ([l,u),v) be the pair in
        {([l1,u1),v1) in pairs[K] | l1 ≤ n < u1}
        with maximal l1
    return v
```

In [McCreight 82], McCreight presents a data structure, called a *balanced priority search tree*, for representing a set of (x,y) pairs of integers, and algorithms for the following operations:

```
InsertPair(x,y):
        add (x,y) to the set D
        O(lg n)

DeletePair(x,y):
        remove (x,y) from the set D
        O(lg n)

MinXInRectangle(x0,x1,y1):
        find a pair in {(x,y) in D | x0≤x≤x1 & y≤y1}
        whose x component is minimal
        O(lg n)

MaxXInRectangle(x0,x1,y1):
        find a pair in {(x,y) in D | x0≤x≤x1 & y≤y1}
        whose x component is maximal
        O(lg n)

MinYInRange(x0,x1):
        find a pair in {(x,y) in D | x0≤x≤x1}
        whose y component is minimal
        O(lg n)

EnumerateRectangle(x0,x1,y1):
        enumerate all {(x,y) in D | x0≤x≤x1 & y≤y1}
        O(|result| + lg n)
```

Now the data structure we want is a set of [L,U] intervals, with operations

```
AddInterval(L,U):
        add an interval [L,U] to the set

GetInterval(x):
        find an interval in {[L,U] | L≤x≤U}
        with maximal L
```

We can implement this with a priority search tree, by defining

```
AddInterval(L,U) =
        InsertPair(-L,-U).

GetInterval(x) =
        [-u,-v] where (u,v) = MinXInRectangle(-x,oo,-x)
```

To see that this does what we want,

```
MinXInRectangle(-x,oo,-x)
    =   the (-L,-U) in {(-L,-U) in D | -x≤-L≤oo & -U≤-x}
        with minimal -L
    =   the (-L,-U) in {(-L,-U) in D : L≤x≤U}
        with maximal L

So
GetInterval(x) =
        the [L,U] | x in [L,U] with maximal L, as required.
```

A fortunate consequence of this definition is that the pairs stored in the priority search tree have distinct first components, which simplifies the algorithms.

McCreight presents two versions of priority search trees, *radix* priority search trees,

which depend on the fact that the pairs are pairs of integers, and do arithmetic on the numbers, and *balanced* priority search trees, which depend only on the order of the keys. We cannot use radix trees, because we want to change the numbers.

McCreight suggests implementing a balanced priority search tree as a linked tree:

```
type
    BPSTptr = ^BSPTrec;
    BPSTrec = record
        p, q: Pair;
        validP, duplQ: boolean;
        left, right: BPSTptr;
        balance: integer;
    end;
```

where Pair is the data type of a pair of keys. The data structure of this chapter associates the numbers with a node in the context tree, so we can implement Pair simply as a pointer to a context tree node. While the associated numbers do change, the ordering between nodes does not.

The combination of these two data structures yields a representation for tree-structured multiple context data bases with the following important properties:

- The cost of a change or lookup does not depend on what has happened to any other key or on the total number of contexts.
- The cost of creating a new context is O(1).
- The cost of change(K,C,V) or lookup(K,C) is O(lgN) where N is the number of contexts in which K has been explicitly defined.

7.5. Nearest Common Ancestors

McCreight's priority search trees can also be used to solve the "nearest common ancestor" problem in logarithmic time. The <x,y> pair we store for a node n in the tree is <Ord(Yds(n)),Ord(n)>.

Suppose we are given two nodes d1 and d2. Without loss of generality, suppose that Ord(d1) $\leq$ Ord(d2). We want to find the If we are given two nodes d1 and d2, we want to find that node n in the tree such that

```
Ord(d2) ≤ Ord(Yds(n)) &
Ord(n) ≤ Ord(d1)
```

with minimal Ord(Yds(n)). But this is precisely

```
MinXInRectangle(Ord(d2), oo, Ord(d1))
```

As before, the actual Ord() values don't matter, only the relative ordering of the nodes, so the priority search tree is not disturbed by dynamic renumbering.

The price of being able to implement comtype() in logarithmic time is that it takes logarithmic time to add a node to the tree, whereas the basic data structure takes unit expected time.

7.6. Summary.

I have described an efficient method for handling simple type trees. This method does not generalise to tangled hierarchies, but chapter 4 described how multiple type systems can be used.

The ability to extend the object taxonomy during a consultation is important for a program like ASA. I was able to produce a dynamic version of MECHO's type system, but updates were very expensive, and MECHO's type system does not provide unit cost subtype tests.

The type system can be combined with a data structure due to McCreight to yield an efficient data structure for multiple context data bases.

Chapter 8
Summary And Conclusions.

8.1. Main Contributions of this Work.

The immediate goals of my research were

- to discover what kinds of knowledge
- and what kinds of reasoning are needed to work out *valid* analyses for *simple* statistical experiments,
- to determine what AI techniques are needed if a computer program is to perform this task,
- and to explore techniques for efficient logic-based expert systems generally.

This volume concentrates on knowledge representation issues. The main points of the text are:

- Chapter 2 presented an approach to classifying measurements. This approach is related to dimensional analysis, but goes further. I maintain that this classification is new and is better suited for non-statisticians than other classifications now in use. For statisticians, the importance of this chapter is that the approach is new, and the catalogue of value spaces can easily be extended without modifying the foundational ideas.
- Chapter 3 presented a notation for simple experiments. Perhaps more importantly, it presented a methodology for extending the notation. Both are new. The notation needs to be extended to handle sampling methods; some work on this has been done but is not reported here for reasons of space. It also needs to be extended to handle time series; that will require extending the methodology. The notation is useful for computers. For statisticians, the importance may be that the notation is usable by people. Trying to write down the formula for an experiment is an extremely useful way of ensuring that you haven't forgotten to ask the right "silly questions".
- Chapter 4 presented a new *kind* of type system, called "descriptions". I argued that vague descriptions are a better way to handle some kinds of uncertain

reasoning than certainty factors. The chapter explained how descriptions can support negation and inheritance. The whole of the chapter is original.

- Chapter 5 presented Plotkin's learning algorithm. This has been known for years. Its relevance to this work is that Plotkin found that the use of lattices as description spaces followed naturally from a desire that descriptions should be learnable. There is original material in this chapter too: although the incorporation of DD rules into the task does not change the *formal* nature of the learning algorithm, it does change its appearance, and the fact that the focussing algorithm can take background information into account has not been reported before. Other authors than Plotkin have presented the algorithm as working with sets of trees; while attention was directed to this special case the representational power of lattices went un-noticed.

- Chapter 6 presented a fast algorithm for finding fixed points. Although this algorithm is closely related to the LINCLOSURE algorithm and to McAllester's propositional constraint propagation (which I believe were independently derived), the generalised algorithm and the demonstration of its linearity are original. This algorithm is relevant to descriptions, to belief revision, and to the analysis of logic programs.

- Chapter 7 presented a dynamic data structure for simple type trees which enables type inferences to be reached rapidly. A version of this data structure was developed at UBC, but that was a *static* structure. If a client is to be allowed to extend the description space during the consultation, a dynamic system is needed. While chapter 5 argued that a simple type tree is inadequate in general (and chapter 2 showed that it is inadequate for Statistics), a product of simple type trees is often useful. When this data structure is combined with McCreight's BSTs, a fast data structure for Conniver-style data bases results. Both the type tree and the multiple context data base data structures are original.

The basic assumption behind all of this research is that first order predicate calculus is a good tool for developing Expert Systems, and that a good approach is to split off special kinds of axioms which can be processed with particular efficiency (such as descriptions and dependencies), but to retain their logical semantics. For an explanation of how this splitting can be done while retaining completeness, see [Stickel 83]. Thus the *language* of first order logic does not entail a commitment to the *machinery* of resolution.

8.2. ABASE

This volume does not describe the ASA program in full. It omits, for example, any discussion of goals or statistical methods. Nor does it describe the ABASE shell in full. There are many interesting topics, such as the variant of Earley deduction I use, the way negation is handled, the relation between the "meta-level" annotations I copied from MECHO and relational data base theory, dependency maintenance/belief revision, natural language output, and rules with uncertainties which could have been discussed.

Each of the topics summarised in this section was actually studied as part of my research. They are not described in the main body of the text because they are not directly relevant to the "knowledge representation and efficient algorithms for it" theme.

8.2.1. Earley Deduction

In 1975, David Warren described a family of proof procedures for Horn clauses which he called "Earley Deduction", as the proof procedures were explicitly modelled on Jay Earley's [Earley 70] famous parsing algorithm. The most recent description of the family is in [Pereira & Warren 83].

What attracted me to this method was that it was not a chronological backtracking procedure such as is built into the programming language Prolog. That approach is fine for programming, but very bad for interactive Expert Systems. The data structures of the Earley Deduction family seemed well suited to dependency maintenance, and the fact that a set of goals is explicitly maintained opened up the possibility of having a "conversation manager" module which would pick questions to ask the client so as to keep the conversation coherent.

My version of the method uses "silly" filtering [Bundy, Byrd & Mellish 82] and dynamic goal ordering, and builds in several predicate properties such as functional dependencies. The current "conversation manager" is very simple: the Earley Deduction chains along until it can't get any further without asking a question. At this point it has a pool of questions which it would be useful to ask. The conversation manager tries to stick to the same topic. For example, if we have been asking about the description, properties, or components of a particular entity, it prefers to ask another question about the same entity. The usual "how" and "why" questions are supported, as are "help" questions to ask about words in ABASE's questions. The client can volunteer information.

8.2.2. Negation

Negation is handled four ways.

- First, de Morgan's laws are used to push negation down to the goal level.
- Second, it is often possible to rename a set of clauses (by moving the negation sign into the predicate name, as it were) so that the resulting set is Horn. See [Lewis 78, Meltzer 66]. I have developed a fast algorithm for this.
- Third, the description system can be used. We can show that an entity does not satisfy one description by showing that the description it is known to satisfy is incompatible with the first one. A special case of this is the practice of adding an extra argument to a predicate. For example, instead of having a predicate

  ```
  british_citizen(Whoever)
  ```

  we might have a two-place predicate

  ```
  british_citizen(Whoever, yes/no)
  ```

  which is declared to be functional in its first argument. Adding a yes/no argument to a predicate is something of a hack, but one can often turn such a predicate into a perfectly respectable description. Assuming that no-one can be a citizen of two countries (yes, I know this is false) we could have a citizenship aspect and ask whether

  ```
  the citizenship of Whoever is british
  ```

- Finally, it is sometimes appropriate to make the closed world assumption. This is particularly appropriate for tables which are built into the rule set. ABASE does permit the client to say that a question has no more answers. But that facility is not currently used to support negation as failure, just to stop ABASE asking the same question again. The reason why the closed world assumption is not allowed for user-supplied data is that allowing it would make dependency maintenance much more complicated. With the other methods, ASA did not need negation on ordinary predicates.

8.2.3. Annotations and Dependencies

An important feature of MECHO was that the inference engine was basically just a depth-first backwards-chaining interpreter like Prolog. But it was controlled by annotations stating that this predicate satisfied such a functional dependency, that predicate was symmetric, another predicate always had a solution, and so on.

Many of these annotations were equivalent to ordinary first-order sentences which MECHO was otherwise unable to express. Most of them, indeed, are embedded implica-

tional dependencies (see [Maier 83]). ABASE can express the dependencies whose translations do not involve equality directly, and because it uses Earley deduction, such rules cause it no special difficulties. Rules (such as functional dependencies) whose translations do involve equality cannot be expressed directly in ABASE, as ABASE does not handle equality at all.

A lot is known about various kinds of dependencies, including methods for computing closures (all the dependencies in a certain family which follow from the ones the knowledge engineer supplied), and methods for finding so-called Armstrong relations. An Armstrong relation for a set of dependencies is an instance of that relation which satisfies all the dependencies in the given family which are logically entailed by the given set, and no others. Armstrong relations are useful for debugging one's annotations.

The availability of methods for computing closures means that all the dependencies a given predicate satisfies can be computed at "compile time". So the inference engine only has to be able to check or exploit single dependencies, and need not do any inference at "run" time to decide which dependencies to check. We can do this without losing completeness, because the closure methods are complete in their own clearly delimited domains.

A failing of MECHO was that it neither precomputed all dependencies nor inferred dependencies at run time, but only checked and exploited the dependencies explicitly stated by the knowledge engineer, and didn't always check or exploit them.

8.2.4. Dependency Maintenance

Dependency maintenance in ABASE is based on [McAllester 78]'s "Propositional Constraint Propagation" restricted to propositional Horn clauses (because the logical rules ABASE works with are translated to Horn clauses). The reason for that is that McAllester's was the first paper on dependency maintenance that I could understand.

In fact the restriction of McAllester's method to propositional Horn clauses is precisely the linear time algorithm for finding minimal models presented in Chapter 5. I did not realise that until I designed the generalised LINCLOSURE algorithm presented there, which was consciously based on LINCLOSURE. It is easy to show (though McAllester does not do this) that the sets of propositional clauses on which his algorithm terminates are precisely the sets which are renamable-Horn.

8.2.5. Natural Language Output

Natural language input is a good idea when the client and consultant share a common jargon. But in Statistics consulting, this is rarely the case. It is also a good idea when the client has a good enough understanding of the problem to volunteer useful information. Again, in Statistics consulting the client's idea of what is important/relevant information and the statistician's idea all too seldom coincide.

When I started work on ASA, Rob Milne was working on the Marcus-style parser used in MECHO [Milne 80a, Milne 80b, Milne 83], an early version of CHAT [Dahl & Sambuc 76] was running, and CHAT-80 [Warren & Pereira 81] was under development. Also, I was familiar with Definite Clause Grammars [Pereira & Warren 78]. So I expected that if I found a use for natural language input, I would either be able to use someone else's parser, or to write an application-specific parser without too much effort. Since then, Michael McCord has published a comparable parser in the AI journal [McCord 82], and I have developed a variant of Definite Clause Grammars based on Generalised Phrase Structure Grammars [Gazdar & Pullum 85]. So my failure to include natural language *input* in ABASE was not for lack of opportunity.

Having used these parsers, I am convinced that for the ordinary client a computer-directed dialogue with short answers from the client (names of entities, some descriptions, menu selections) is *less* confusing than natural language input. When a parser fails to handle your sentences, it is very difficult to work out what went wrong. Also, many potential clients these days are unlikely to be used to thinking of language as something that can be manipulated, and to have difficulty with the very idea of paraphrasing a statement using different syntax.

None of these objections applies to natural language *output*. The knowledge representation language of ABASE is very simple, there are

- entities - if the user or knowledge engineer created them, they correspond to proper nouns, otherwise they correspond to definite noun phrases
- descriptions - these come out as adjectives and so on.
- predications - these correspond to simple sentences in the basic form

```
subject verb [indirect] [direct] pp...
```

Simple sentences are transformed to bring the most interesting entities to the front.

If the knowledge engineer provides a set of templates for the predicates and descriptions, there is a module NEWSAY which is able to generate simple natural language output. For example, if asked to describe Hannah, NEWSAY might report

```
Hannah is a Samoyed.
It is the mother of Samuel.
```

If asked to describe Samuel, NEWSAY might report that

```
Samuel is a dog.
Its mother is Hannah.
```

Note that the same mother(Hannah,Samuel) fact was expressed differently depending on which one we are most interested in. There is a small set of rules for bringing a particular noun phrase to the front, that is all.

NEWSAY does not handle gender. That shows up in this example, but in the actual Statistics consulting domain all the entities we are talking about are abstract and gender is not an issue.

A conjunction feature, which would have combined sentences which differed only in the last noun phrase, was designed but never coded. Relative clauses are generated, but only for entities which were created by inference rules. They are *not* created by combining sentences, but are generated directly as noun phrases from the facts the nameless entities were created for. No other method of combining or nesting sentences was written or intended.

Indeed, the whole sentence generation process was so simplified and constrained by ABASE's knowledge representation scheme that it was possible to implement NEWSAY as a definite clause grammar running "backwards".

This approach is glaringly inadequate for generating coherent paragraphs. But for a simple conversational interface it works quite well.

8.2.6. Uncertainties

My research on logic programs with uncertainties was inspired by [Shapiro 83]. That paper takes uncertainties as elements of the total order (0,1]. My approach took uncertainties as elements of any specified complete lattice.

If the lattice used as a certainty space has finite depth, the finite fixed point method developed in Chapter 4 can be used to compute or update certainties in time linear in the size of the current partial proof.

ASA does not use certainty factors at all, so ABASE has no provision for them. Indeed, in Chapter 2, I argue that ascribing certainty factors to whole predications is unwise, and that if anything, uncertainty belongs with the *arguments* of predications.

Since the logic programs with uncertainties study is not relevant to ASA, it is not included in this volume.

8.3. Directions for Future Research

Recall that the goals of my research were to explore representational and inference issues, and to look for efficient algorithms and data structures. It would have been nice to build all the ideas into one program, but since the production of a deliverable program was not the point, I never let this interfere with exploring a new idea.

At no time have all the pieces of ABASE worked together. There is a version of ABASE which really does run rule bases stated in Horn clauses, using Earley Deduction, but it does not use the current description system. The conversation manager and the explanation and help systems work in it. There is an implementation of the current description system, but it doesn't work with anything else. There is a version of NEWSAY, but it works with the previous type system. There is a program which computes closures of annotations, and its results can be fed back into ABASE, but ABASE doesn't understand all the annotations that this program can handle. There is a dependency system in the running ABASE, but it doesn't use my new algorithm, though that has been tested.

As ABASE has never all worked at once, many of the rules in ASA have only been exercised in hand simulations, mainly the model selection rules and some of the transformation selection rules. All of the method description rules have been exercised and work. But ASA has never been in a state where real clients could use it, and has only been tried on textbook examples.

So one direction for future research would be to produce a new version of ABASE with all of the components working together, and to extend the ASA rule-base.

One of the principal reasons why ABASE was developed as several incompatible pieces was that there was never a statistician involved, nor any likely client. It was always more rewarding to look for a better algorithm than to integrate the existing algorithms into a complete program. Statisticians do not need a consulting program such as ASA was intended to be. But the craft of Statistics is *not* the craft of running statistical packages, but the craft of designing experiments and *explaining* the results to *people*.

Accordingly, another direction for future research would be to produce an "Animated CodeBook". That is, to produce a statistical computing environment where the "package" is told the structure of the experiment and the nature of the measurements. This would be useful when

- several statisticians are working on the same experiment. With an Animated CodeBook it would be easier to keep track of what several hundred variables mean. For example, a CodeBook user could ask "which variables measure political affiliations", and the CodeBook could answer.

- a report is being written. The CodeBook could keep track of how variables were derived, what the terms in a model meant, and could maintain cross references to related experiments.
- a statistician is exploring the data. The CodeBook could suggest graphs which might be interesting, transformations which might be useful, and could warn when a method is about to be used inappropriately. The point of the CodeBook would be to ensure that the obvious was not overlooked.

The Animated CodeBook project would have to extend the structural notation to handle sampling, to explicitly record which processes were used to take measurements, and to handle classical experiment structures, but this would fit very well into the framework I have presented, and some of it has already been thought about.

It would be interesting to build the fast data structure for multiple context data bases outlined in chapter 7 into a logic programming language; Prolog/KR and LM-Prolog have multiple world data bases, but this would be better.

Can the fast type-tree algorithm be extended to handle partial orders? Or is it possible to prove that it cannot? Either would be a useful result.

In section 1.1.2 I presented an "algorithm" for doing science. Step [10], "use the collected data to refine the model, or to suggest a replacement for it", was not part of this research. It should be possible to do something here. For example, here are some heuristics:

- look for a correlation between the discrepancies and some other factor which has not entered the model yet. For example, check for a temporal trend.
- test whether it is possible that the level of some factor at the active site is different from the level measured or applied. For example, in an agricultural experiment, it is worth measuring the amount of fertiliser which reaches root level as well as reporting the amount which was applied.
- consider breaking a monomial (c.X^a.Y^b) into a polynomial (c1.X^a1.Y^b1 + c2.X^a2.Y^b2)

This obviously ties back into Machine Learning. What do you do when the rule you have learned perfectly fits the training set but turns out to be wrong?

8.4. Conclusion

I believe that a knowledge representation language should have firm semantic foundations. For if there is no objective way of determining what a notation *means*, how can I tell whether what I have written says what I think it does, and how can it make sense to ask whether a program intended to support the notation is functioning correctly?

First-order logic is a good place to start. When one identifies common patterns of reasoning such as types, part/whole, ordering, and so on, one can call on work such as [Stickel 83] to handle these specialised inferences without destroying the advantages of logic.

A particularly important advantage of first-order logic which has been largely overlooked by the AI community is that for many years people in the data processing community have been trying to model large real-world systems. The data base community is as deeply concerned with "knowledge representation" as the AI community, and their methods [Maier 83] are intimately related to first-order logic. The modelling disciplines developed by the data base community can be applied to AI problems.

One common pattern of reasoning concerns descriptions. I have developed a lattice-based approach to descriptions which is reasonably general and has a firm mathematical basis.

I have presented algorithms and data structures for performing certain kinds of inferences efficiently.

Bibliography

[Beeri 79] Beeri, C., and Bernstein, P.A.
Computational Problems Related to the Design of Normal Form Relational Schemas.
ACM Transactions on Database Systems 4(1):30-59, March, 1979.

[Birkhoff 67] Birkhoff, G.
Colloquium Publications. Volume XXV: *Lattice Theory.*
A.M.S., 1967.

[Brachman 85] Brachman, Ronald J.
"I Lied about the Trees" Or, Defaults and Definitions in Knowledge Representation.
The AI Magazine (3):80-93, Fall, 1985.

[Bundy & Silver 82]
Bundy, A. and Silver, B.
A critical survey of rule learning programs.
In *Proceedings of ECAI-82*, pages 150-157. European Conference on Artificial Intelligence, 1982.
Also available from Edinburgh as DAI Research Paper No. 169.

[Bundy *et al* 76a] Bundy, A., Luger, G., Stone, M. & Welhem, R.
Mecho: Year One.
In Brady, M. (editor), *Procs of AISB2*, pages pp94-103. Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1976.
Also available from Edinburgh as Research Paper 22.

[Bundy *et al* 76b] Bundy, A., Luger, G., Stone, M. & Welham, R.
Mecho: Year One.
Research Paper 22, Dept. of Artificial Intelligence, Edinburgh, 1976.
Also in proceedings of AISB Summer conference.

[Bundy *et al* 77] Bundy, A., Luger, G., Mellish, C. and Palmer, M.
Solving Mechanics Problems: Interim Report to the SRC.
Working Paper 23, Dept. of Artificial Intelligence, Edinburgh, December, 1977.

[Bundy *et al* 79a] Bundy, A., Byrd, L., Luger, G., Mellish, C., Milne, R. and Palmer, M.
Solving Mechanics Problems Using Meta-Level Inference.
In Buchanan, B.G. (editor), *Proceedings of IJCAI-79*, pages 1017-1027. International Joint Conference on Artificial Intelligence, 1979.
Reprinted in 'Expert Systems in the microelectronic age' ed. Michie, D., Edinburgh University Press, 1979. Also available from Edinburgh as DAI Research Paper No. 112.

[Bundy *et al* 79b] Bundy, A., Byrd, L., Luger, G., Mellish, C., Milne, R. and Palmer, M.
Mecho: A program to solve Mechanics problems.
Working Paper 50, Dept. of Artificial Intelligence, Edinburgh, 1979.

[Bundy *et al* 84] Bundy, A., Silver, B. and Plummer, D.
An Analytical Comparison of some Rule Learning Programs.
Research Paper 215, Dept. of Artificial Intelligence, Edinburgh, 1984.
Submitted to Artificial Intelligence Journal. Earlier Version in Procs of Third Annual Technical Conference of the British Computer Society's Expert System Specialist Group, 1983.

[Bundy, Byrd & Mellish 82]
Bundy, A., Byrd, L. and Mellish, C.
Special Purpose, but Domain Independent Inference Mechanisms.
In *Proceedings of ECAI-82*, pages 67-74. European Conference on Artificial Intelligence, 1982.
Also available from Edinburgh as DAI Research Paper No. 179.

[Chen 76] Chen, P.P.
The Entity-Relationship Model: Toward a Unified View of Data.
ACM Transactions on Database Systems 1(1):9-36, 1976.

[Chen 83] Chen, P.P. (editors).
Entity-Relationshop Approach to Information Modeling and Analysis.
North-Holland, 1983.

[Cochran 57] Cochran, W.G. & Cox, G.M.
Probability and Statistics: Experimental Designs, Second Edition.
John Wiley & Sons, 1957.

[Cochran 77] Cochran, W.G.
Sampling Techniques, 3rd Ed.
John Wiley & Sons, 1977.

[Dahl & Sambuc 76]
Dahl, V. and Sambuc, R.
Un systeme de bases de donnees en Logique du Premier Ordre, en vue de sa consultation en langue naturelle.
Rapport, Universite d'Aix Marseille, 1976.

[Duda et at 78] Duda, R.O., Hart, P.E., Nilsson, N.J. and Sutherland, G.L.
Semantic network representations in rule-based inference systems.
In Waterman, D and Hayes-Roth, F. (editor), *Pattern- directed inference systems*, pages 203-221. Academic Press, 1978.

[Earley 70] Earley, J.
An Efficient Context-Free Parsing Algorithm.
Communications of the ACM 13(2):94-102, February, 1970.

[Erceg 76] Erceg, Michael A.
Functions, Equivalence Relations, Quotient Spaces, and Subsets in Fuzzy Set Theory.
Report Series 101, Department of Mathematics, University of Auckland, August, 1976.

[Everitt 77] Everitt, B.S.
The Analysis of Contingency Tables.
Chapman and Hall, 1977.

[Fahlman 79] Fahlman, S.E.
NETL: A system for Representing and Using Real-World Knowledge.
MIT Press, 1979.

[Gazdar & Pullum 85]
Gazdar, G., Klein, E., Pullum, G.K., and Sag, I.A.
Generalized Phrase Structure Grammar.
Harvard University Press, 1985.

[Gierz 80] Gierz, G., Hofmann, K.H., Keimel, K., Lawson, J.D., Mislove, M., Scott, D.S.
A Compendium of Continuous Lattices.
Springer-Verlag, 1980.

[GLIM 78] Baker, R.J. and Nelder, J.A.
The GLIM System Release 3 Manual.
The Numerical Algorithms Group, 1978.

[Gordon 79] Gordon, M.J.C.
The Denotational Description of Programming Languages; An Introduction.
Springer-Verlag, 1979.

[Gordon *et al* 79] Gordon M.J.C., Milner A.J., and Wadsworth C.P.
Lecture Notes in Computer Science. Volume 78: *Edinburgh LCF - A mechanised logic of computation.*
Springer Verlag, 1979.

[Hajek 81] Hajek, Jaroslav.
Sampling from a Finite Population.
Dekker, 1981.

[Hampel 86] Hampel, F.R, Ronchetti, E.M., Rousseeuw, P.J., & Stahel, W.A.
Probability and Mathematical Statistics: *Robust Statistics, the approach based on influence functions.*
John Wiley & Sons, 1986.

[Harbison & Steele 84]
Harbison, S.P., and Steele, G.L.
C: A Reference Manual.
Prentice-Hall, 1984.

[Huber 81] Huber, P.J.
Probability and Mathematical Statistics: *Robust Statistics.*
John Wiley & Sons, 1981.

[Kernighan 78] Kernighan, B.W. and Ritchie, D.M.
The C Programming Language.
Prentice-Hall, 1978.

[Knuth 73] Knuth, D.E.
Fundamental Algorithms, 2nd Edition.
Addison-Wesley, 1973.

[Kuhn 70] Kuhn, T.S.
The Structure of Scientific Revolutions, 2nd ed.
University of Chicago Press, 1970.

[Langley *et al* 83] Langley, P., Zytkow, J. Simon, H. & Bradshaw, G.
Mechanisms for Qualitative and Quantitative Discovery.
In Michalski, R.S (editor), *Proceedings of the International Machine Learning Workshop*, pages 121-132. University of Illinois, June, 1983.

[Lewis 78] Lewis, Harry B.
Renaming a Set of Clauses as a Horn Set.
Journal of the ACM 25(1):134-135, January, 1978.

[Lloyd 84] Lloyd, J.W.
Foundations of Logic Programming.
Springer-Verlag, 1984.

[Maier 83] Maier, David.
The Theory of Relational Databases.
Computer Science Press, Inc., 11 Taft Court, Rockville, Maryland 20850, 1983.
highly recommended!

[Marples 74] Marples, D.
Argument and technique in the solution of problems in Mechanics and Electricity.
CUED/C-Educ/TRI, Dept. of Engineering, Cambridge, England, 1974.

[McAllester 78] McAllester, D.A.
A Three Valued Truth Maintenance System.
AI Lab Memo 473, Massachusetts Institute of Technology, May, 1978.
Describes propositional constraint propagation.

[McCarthy 80] McCarthy, J.
Circumscription - A Form of Non-Monotonic Reasoning.
Artificial Intelligence (13):27-39, 1980.

[McCord 82] McCord, M.C.
Using Slots and Modifiers in Logic Grammars for Natural Language.
Artificial Intelligence 18(3):327-367, 1982.

[McCreight 82] McCreight, Edward M.
Priority Search Trees.
Research Paper CSL-81-5, XEROX, January, 1982.
Also appears in SIAM J. Computing.

[Mellish 85] Mellish, C.S.
Global Optimisations for a Prolog Compiler.
J. Logic Programming 2(1):43-66, April, 1985.

[Meltzer 66] Meltzer, B.
Theorem-proving for computers: Some results on resolution and renaming.
Computer Journal 8(4):341-343, January, 1966.

[Milne 80a] Milne, R.
Using Determinism to Predict Garden Paths.
In *AISB 80 Conference Proceedings.* AISB, 1980.
Also available from Edinburgh as Research Paper 142.

[Milne 80b] Milne, R.
Parsing Against Lexical Ambiguity.
In *Proceedings for COLING 80.* , 1980.
Also available from Edinburgh as Research Paper 144.

[Milne 83] Milne, R.
Resolving Lexical Ambiguity In A Deterministic Parser.
PhD thesis, Dept. of Artificial Intelligence, Edinburgh, 1983.

[Mosteller 78] Mosteller, F. and Tukey, J.W.
Data Analysis and Regression.
Addison-Wesley, 1978.

[O'Keefe 83a] O'Keefe, R.A.
Concept Formation From Very Large Training Sets.
In Bundy, A. (editor), *Proceedings of the Eighth IJCAI*, pages 479-481. International Joint Conference on Artificial Intelligence, 1983.
Also available from Edinburgh as Research Paper 192.

[O'Keefe 83b] O'Keefe, R.A.
Concept Formation With Numeric Attributes.
Working Paper 154, Dept. of Artificial Intelligence, Edinburgh, October, 1983.

[O'Keefe 85] O'Keefe, R.A.
An Algebra for Constructing Logic Programs.
In Doug DeGroot (editor), *1985 International Symposium on Logic Programming.* IEEE Computer Society Press, 1985.
Also available as an Edinburgh DAI Research Report.

[Pereira & Warren 78]
Pereira, F.C.N. and Warren, D.H.D.
Definite clause grammars compared with augmented transition networks .
Dept. of Artificial Intelligence, Edinburgh, 1978.
Research Report 58.

[Pereira & Warren 83]
Pereira, F.C.N., and Warren, D.H.D.
Parsing as Deduction.
Technical Note 295, SRI International, June, 1983.

[Peters 83] Peters, R. H.
The Ecological Implications of Body Size.
Cambridge Studies in Ecology; Cambridge University Press, 1983.

[Plotkin 69] Plotkin, G.
A note on inductive generalization.
In Michie, D and Meltzer, B (editors), *Machine Intelligence 5*, pages 153-164. Edinburgh University Press, 1969.

[Plotkin 70] Plotkin, G.
Inductive Generalisation.
PhD thesis, Dept. of Artificial Intelligence, Edinburgh, 1970.

[Plotkin 71] Plotkin, G.
A further note on inductive generalization.
In Michie, D and Meltzer, B (editors), *Machine Intelligence 6*, pages 101-126. Edinburgh University Press, 1971.

[Plummer 83] Plummer, D.
Two techniques for Inductive Learning: A Comparison.
Research Paper 186, Dept. of Artificial Intelligence, Edinburgh, March, 1983.

[Quinlan 79] Quinlan, J.R.
Discovering Rules by Induction from Large Collections of Examples.
In Michie, D. (editor), *Expert Systems in the Micro-Electronic Age*, pages 168-201. Edinburgh University Press, Edinburgh University Press, 1979.

[Ramamohanorao 86]
Ramamohanorao, K. & Shepherd, J.
A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases.
In Shapiro, E. (editors), *Third International Conference on Logic Programming*, pages 569-576. Springer-Verlag, 1986.
Lecture Notes in Computer Science No. 225.

[Reiter 81]
Reiter, R.
On the Integrity of Typed First-Order Data Bases.
Advances in Data Base Theory -- Volume 1.
Plenum Press, 1981, pages 137-158.
separation into clauses and type-data-base, types are unary preds.

[--- 82]
Forgy, Charles L.
Rete: a Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.
Artificial Intelligence , September, 1982.

[Schmolze 83]
Schmolze, J.G. & Lipkis, T.A.
Classification in the KL-ONE Knowledge Representation System.
In *Proceedings of IJCAI-83*, pages 330-332. International Joint Conference on Artificial Intelligence, 1983.

[Sedgwick 83]
Sedgwick, R.
Algorithms.
Addison-Wesley, 1983.

[Shapiro 83]
Shapiro, E.Y.
Logic Programs With Uncertainties: A Tool For Implementing Rule-Based Systems.
In *Procs. of IJCAI-83*, pages 529-532. International Joint Conference on Artificial Intelligence, 1983.

[Shortliffe 76]
Shortliffe, E.H.
Computer-based medical consultations: MYCIN.
North Holland, 1976.

[Siegel 56]
Siegel, Sidney.
Nonparametric Statistics for the Behavioral Sciences.
McGraw-Hill, 1956.

[SPSS 75]
Nie, Hull, Jenkins, Steinbrenner, and Bent.
Statistical Package for the Social Sciences, 2nd Ed.
McGraw-Hill, 1975.

[Stevens 46]
Stevens, S. S.
On the Theory of Scales of Measurement.
Science 103:677-680, 1946.

[Stickel 83]
Stickel, Mark E.
Theory Resolution: Building in non-Equational Theories.
Technical Note 286, SRI International, May, 1983.

[Velleman 81]
Velleman, P.F., and Hoaglin, D.C.
Applications, Basics, and Computing of Exploratory Data Analysis.
Duxbury, 1981.

[Warren 74] Warren, D.H.D.
WARPLAN: A system for generating plans.
DCL Memo 76, Dept. of Artificial Intelligence, Edinburgh, 1974.

[Warren & Pereira 81]
Warren, D.H.D. and Pereira, F.C.N.
An efficient easily adaptable system for interpreting natural language queries.
Research Paper 155, Dept. of Artificial Intelligence, Edinburgh, 1981.
[Submitted to American Journal of Computational Linguistics].

[Warshall 62] Warshall, S.
An Algorithm for Computing the Transitive Closure.
Journal of the ACM , 1962.

[Weisberg 80] Weisberg, Sanford.
Applied Linear Regression.
John Wiley & Sons, 1980.

[Winston 75] Winston, P.
Learning structural descriptions from examples.
In Winston, P.H. (editor), *The psychology of computer vision.* McGraw Hill, 1975.

[Young *et al* 77] Young, R.M., Plotkin, G.D. and Linz, R.F.
Analysis of an extended concept-learning task.
In Reddy, R. (editor), *Proceedings of IJCAI-77*, pages 285. International Joint Conference on Artificial Intelligence, 1977.

Appendix A
Experiment Structure in ASA.

Chapter 3 described a notation for experiment structure which I have found convenient when writing experiment descriptions out by hand. It is not convenient to use that notation in ASA.

Instead of using a formula, each sub-formula is given a name and represented as an "entity" (of type "step"). This allows us to represent steps about which we have as yet little information (vague steps), and avoids the problems associated with compound names.

There are several closely related entities we could represent: an experiment step, the stream of subjects which enter it, a typical member of that stream. Because streams are set-like objects, and because ABASE does not handle set-like objects well (this has proven to be a limitation in several areas), I chose not to represent streams at all, but to distribute their properties over the corresponding steps and typical subjects.

There is no automatic connection between the typical subject of one step and the typical subject of another step, though the rule that introduces a step can of course state that it has the same typical subject as another step. The reason for this is that a typical subject really represents a subject plus his history; so that identifying "the typical subject before he has had his tonsils removed" and "the typical subject after he has had his tonsils removed" is not always a good idea. Conditional description-to-description rules provide the means whereby properties can be transferred from one typical subject to another, and a typical subject in this representation is little more than a bundle of properties anyway.

The hierarchy of experiment steps is currently

```
step                        -- "vague"
    then
    separate
    assign
    select
    disperse
    group
```

and the hierarchy of variables discussed in chapter 3 is

```
variable
    observation
    treatment
    identifier
        own_name
        serial
        rank_order
            rank_random
            rank_serial
```

Every step has the following attributes:

size(Step, Size) Size is an interval, which starts out at the unknown value for intervals of 0..infinity. It is the number of subjects which passed through this step, or which are to pass through it if the experiment has yet to be performed.

typical_subject(Step, Subject)

Subject is a (prototypical) physical object name. The D that appears in the signatures described in chapter 3 is actually the description of this individual. Many steps may have the same typical subject, but it is up to the rule which creates a step whether it has a new typical subject or not.

known_at(Step, Variable)

Variable is one of the variables that have been applied or measured by the time we get to this step. The B set that appears in a signature is the set of these. It is in fact calculated from the determined_in attribute.

determined_in(Step, Variable)

Variable is one of the variables that are applied or measured in this step. This fact is automatically deduced for component identifiers and such. Indeed, if we had the complete experiment structure to start with, we could deduce this from the substeps of this step. But as we start with the vaguest structure we can use, this information has to be obtained initially from the client and then "pushed in" as far as possible.

I am not satisfied with the way that the sets of known and determined variables are represented. The problem is illustrated by "then" steps. Suppose we have (s1 then s2), where the combined step is known to determine variables U,V,W. If we could ask "what subset of {U,V,W} is determined in s1", then we could compute the complement of that and assign it to s2. Instead, we have to ask for each of U,V,W, which step it is determined in. In fact, in order to get a type-like value to discriminate on, it was necessary to break up the notion of which substep a variable is measured in into two predicates:

```
which_substep(Step, Variable, Which) ->
        Step is a then_step &
        Variable is a variable &
        Which is first_or_second.
```

```
first_or_second = {first, second}.

determined_in(Step, Variable) &
which_substep(Step, Variable, first) &
part_one(Step, SubStage1) ->
determined_in(SubStage1, Variable).

determined_in(Step, Variable) &
which_substep(Step, Variable, second) &
part_two(Step, SubStage2) ->
determined_in(SubStage2, Variable).
```

Clumsy as this may be, there is one good outcome: if the client discovers that he has forgotten a variable, we can simply add a fact recording that the variable is measured somewhere in the experiment, and track it to its lair only when we need to. This means that the representation ASA uses is actually more flexible than the notation presented in chapter 3. We do not refine a vague step into a sequence of treatments and observations all at once, but can leave each variable separately vague until we need to know where *that* variable is determined. If ABASE supported sets of entities as values of properties, the notation of chapter 3 could have been represented more concisely and faithfully, but this flexibility would have been lost. In fact, clients often do forget variables, especially when designing an experiment.

The predicates we have for each of the refinements are

part_one(ThenStage, SubStage1)

part_two(ThenStage, SubStage2)

These are the first and second steps of a sequence. The step names are invented by ABASE, not by the client, so we have a rule like

```
ThenStage is a then_step ->
        S1^ part_one(ThenStage, S1) &
        S2^ part_two(ThenStage, S2).

part_one(TS,S1) & known_at(TS,V)-> known_at(S1,V).
part_two(TS,S2) & known_at(TS,V)-> known_at(S2,V).
part_one(TS,S1) & part_two(TS,S2) &
            determined_in(S1,V) -> known_at(S2,V).
```

separate_part(Step, Component, SubStage)

Step is a "separate" step. Component is a "field name", that is, the selector of a class of components, not the name of a typical individual. SubStage is the step to which those components are routed. We might have facts like

```
separate_part(step1, male, step2).
separate_part(step1, female, step3).
typical_individual(step1, couple1).
typical_individual(step2, man1).
typical_individual(step3, woman1).
```

```
man1 is the male of couple1.
woman1 is the female of couple1.
```

assign_component(AssignStage, Component)

Component is a selector name for a has_set component.

assign_var(AssignState, Identifier)

Identifier is an identifier variable, which indicates which branch a particular component went down. It has the status of a treatment in some respects.

assign_part(AssignStage, Value, SubStage)

SubStage is the substep of the 'assign' step corresponding to the specified value of the variable.

select_var(SelectStage, Treatment)

note that each object coming into a 'select' step is routed in its entirety to one of the substeps, so there is no need to relate things in different substeps to a common parent.

select_part(SelectStage, Value, SubStage)

SubStage is the substep of the 'assign' step corresponding to the specified value of the variable.

disperse_component(DisperseStage, Component)

Component is the name of a "has_set" component.

disperse_var(DisperseStage, Identifier)

disperse_part(DisperseStage, SubStage)

a 'disperse' step has only one substep.

group_level(GroupStage, LevelVariable)

LevelVariable is a measurement (it must not be a treatment or identifier) which has already been made on the units. The units will be ranked on that variable.

group_size(GroupStage, Size)

Size is an integer, at least 2. Having been ranked, the units are divided into successive blocks of this size.

group_var(GroupStage, GroupVar)

GroupVar is a an identifier introduced by this step. It is in effect a rank, a count of groups.

group_part(GroupStage, SubStage)

A 'group' step has only one substep. Note that the group step itself is fed a stream of units, and it feeds its substep a stream of sets of units.

With these predicates, we can completely describe the structure of any experiment ASA was designed to handle. This is very far from being all statistical experiments of interest, but it covers a wide range of simple experiments, which people often get wrong. Indeed, many experiments don't need refining past a single vague step, as the variable descriptions are enough to guide the analysis.

A.1. Example

The examples in this section comes from pp 163-165 of [Siegel 56]. That book was the main source of test cases for ASA. The first prototype of ASA was able to tackle 25 of the examples in [Siegel 56]. The discovery of lattice-based descriptions opened the door to tackling them all. One of the things which the first prototype could not represent was composite entities, particularly entities which are really just aggregations (matched samples). The example illustrates how this is done now.

> Suppose we were interested in the influence of interviewer friendliness upon housewives' responses in an opinion survey. We might train an interviewer to conduct three types of interviews:
>
> - interview 1, showing interest, friendliness, and enthusiasm
> - interview 2, showing formality, reserve, and courtesy
> - interview 3, showing disinterest, abruptness, and harsh formality.
>
> The interviewer would be assigned to visit 3 groups of 18 houses, and told to use interview 1 with one group, interview 2 with another, and interview 3 with the third. That is, we would have obtained 18 sets of housewives with 3 matched housewives (equated on relevant variables) in each set. For each set, the 3 members would be randomly assigned to the three conditions (types of interviews). Thus we would have 3 matched samples (k=3) with 18 members in each (N=18). We could then test whether the gross differences between the three styles of interviews influenced the number of "yes" responses given to a particular item by the three matched groups. Using artificial data, a test of this hypothesis follows.

The first thing to note about this is that such an experiment should not be performed. The subjects would have given their consent to the ostensible purpose of the experiment (finding out their opinion) but not to the covert purposes, of which they would not have been informed. Deception of this sort is a regrettable feature of many psychological experiments.

The second thing to note is that I have not altered this text. Yes, a textbook used by generations of social scientists is in fact that confused, and that confusing. This example supports my claim that natural language input is not particularly useful in this area: if a statistician can confuse 18 groups of 3 with 3 groups of 18, what hope is there that a naive client will get it right?

It is important to realise that each group of 3 is cohesive in a way that the the groups of 18 are not.

The third thing to note about this experiment is that we don't really have enough information to do a proper job. We cannot represent the sampling phase, because we do not know what the "relevant" variables were. The text is reasonably clear that the housewives didn't come already grouped in neat sets of 3, but that a sample of 54 housewives was drawn in some fashion and grouped into similar triples. In trying to draw the diagram for this experiment, I found that (a) we aren't told how the set of 54 were chosen, (b) we are not told what the "relevant" variables are, and (c) we are not told how the matching is done. In fact I have never found in any textbook a description of how to form matched groups. Discovering these information gaps is one of the most valuable things about trying to draw experiment diagrams.

For argument's sake, let's suppose that the "relevant variables" were just "age", and that the method of grouping was to sort the housewives by age and take successive blocks of 3. Then we end up with the following formula:

```
1. {sample[54] then
2.  observe['age'] then
3.  group['age'; 3; 'group';
4.      assign[self; style;
5.          1 -> observe['answer'],
6.          2 -> observe['answer'],
7.          3 -> observe['answer']]]
   }(study_population).
```

That is,

1. a sample of 54 housewives is drawn.
2. the age of each housewife is measured.
3. the subjects are grouped into triples of similar age.
4. the triples are broken up again, with each member of a triple being assigned to a different interview style.
5. the subjects assigned to style 1 have their answer recorded. A richer notation than ASA's would indicate the measuring instructments. The apparent identity of the three substeps is due to the suppression of such relevant detail.
6. the subjects assigned to style 2 have their answer recorded.
7. the subjects assigned to style 3 have their answer recorded.

The structure of the data set implied by this formula is

Case number	Age	Group	Style	Answer
1	42	13	3	0
....	...	...	...	...
54	27	2	1	1

(The left-to-right order of the columns is the order in which the variables are determined.) This is, for example, how one would present the data to SPSS. A common source of confusion to users of such packages is the discrepancy between a data set structure like this and the tabular forms used in their textbooks. For example, Siegel presents his artificial data in the form

Triple	Answer1	Answer2	Answer3
1	0	0	0
...	...	...	...
18	1	1	0

Such a restructuring of the data is useful for calculation, but it is not directly related to the structure of the experiment.

The representation in ASA is

```
experiment(step2).

step1 is a sampling.                % line 1.
size(step1, 54..54).
typical_subject(step1, unit1).
unit1 is a housewife.

step2 is a then_step.
typical_subject(step2, unit1).
part_one(step2, step1).
part_two(step2, step3).

step3 is a then_step.
typical_subject(step3, unit1).
part_one(step3, step4).
part_two(step3, step5).

step4 is an observe_step.           % line 2.
typical_subject(step4, unit1).
determined_in(step4, age).
property(age1, age, unit1).
age1 is an observable_property.
the value_space of age1 is physprop(time,housewife).
measures(age, age1).
age is an observation.
the value_space of age is physprop(time,housewife).
```

```
step5 is a group_step.          % line 3.
typical_subject(step5, unit1).
group_level(step5, age).
group_size(step5, 3).
group_var(step5, group).
group is an identifier.
the value_space of group is count(unit2).
group_part(step5, step6).

unit2 is a set(housewife, 3..3).

step6 is an assign_step.        % line 4.
size(step6, 18..18).
typical_subject(step6, unit2).
assign_component(step6, self).
assign_var(step6, style).
style is an identifier.
the value_space of style is class(interview,style,[is1,is2,is3]).
assign_part(step6, is1, step7).
assign_part(step6, is2, step8).
assign_part(step6, is3, step9).

element(unit3, unit2).
element(unit4, unit2).
element(unit5, unit2).
unit3 is a housewife.
unit4 is a housewife.
unit5 is a housewife.

step7 is an observe_step.       % line 5.
typical_subject(step7, unit3).
determined_in(step7, answer).
answer is a class(question,reply,[yes,no]).

step8 is an observe_step.       % line 6.
typical_subject(step8, unit4).
determined_in(step8, answer).

step9 is an observe_step.       % line 7.
typical_subject(step9, unit5).
determined_in(step9, answer).
```

This is the basic skeleton. A lot of these facts are "inherited" from one node to another. For example, steps 7, 8, and 9 inherit the size 18 from step6.

The breakthrough was being able to say that unit2 is a set of between 3 and 3 housewives. This hasn't been discussed before, as it is an object type, not a value space or step type. Basically, if D is any object description, and [L,U] is any interval of integers,

```
set(D, [L,U])
```

is an object description, describing all sets of between L and U objects comprised of objects satisfying description D. The lattice rules are

```
set(D1,I1)\/set(D2,I2) = set(D1\/D2,I1\/I2)

set(D1,I1)/\set(D2,I2) = set(D1/\D2,I1/\I2)

set(D, {}) = 0.
```

"element" is like "component", except that it refers to sets. The "unique name assumption" means that ABASE assumes that units 3, 4, and 5 are distinct elements of unit 2. In ASA, this true by construction. Some of the details have been omitted.

For programmed inference, the representation as ABASE facts and types is superior. For human use, the functional notation is superior.

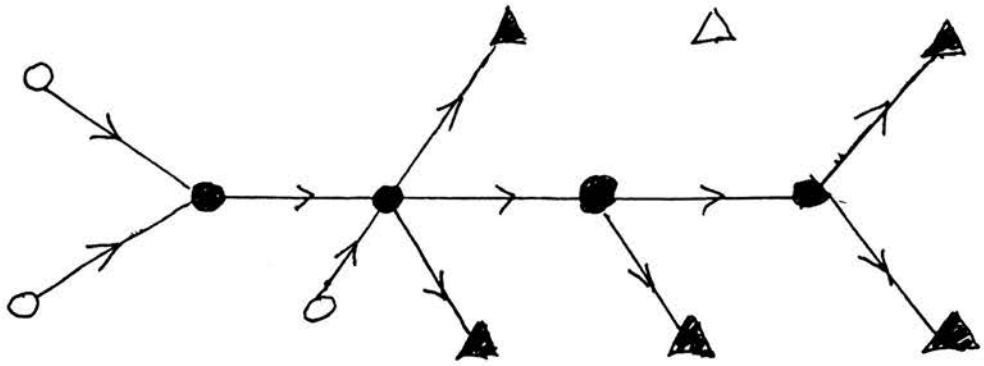
Appendix B
MultiMetic Scales

As a result of my work on ASA, I have identified a new member of the family of partially ordered scales. It appears that many variables which are customarily analysed as if they were measured on an ordinal scale actually belong to this new scale. I propose a method for analysing data belonging to multi-metic scales.

B.1. Partially Ordered Scales.

A partially ordered scale is a set of values and a binary relation "<" which is a partial order. When there are few values, it can be useful to draw a graph showing the structure of the scale, with a node for each value in the set and an arc directed from node X to node Y precisely when X < Y and there is no Z such that X < Z and Z < Y. I have found it useful to adopt the following convention: a node with no arcs leaving it (a maximal value) is drawn as a triangle, others as circles, a node with no arcs entering it (a minimal value) is an outline, others are solid. Figure 1 shows a typical partial order.

Figure 1. A Nondescript Partial Order.



B.2. Bimetic and Multimetic Scales.

Both nominal scales (Figure 2) and ordinal scales (Figure 3) are special cases of partially ordered scales.

Figure 2. A Nominal Scale.

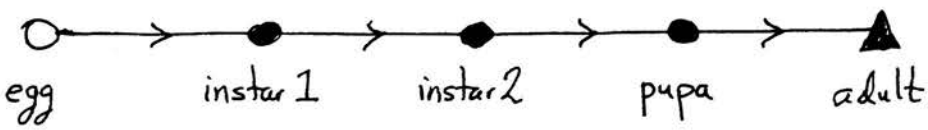


Figure 3. An Ordinal [Unimetic] Scale.

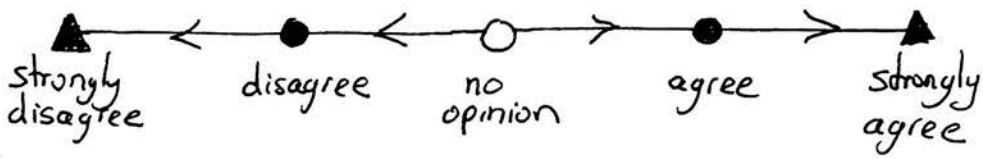


Figure 4. A Bimetic Scale.

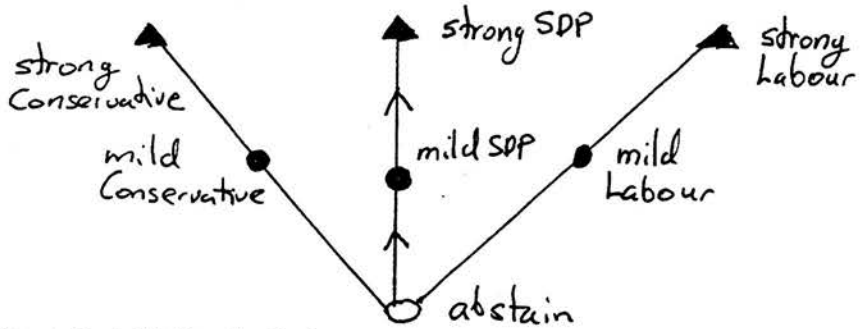


Figure 5. A Multimetic Scale.

Figures 4 and 5, however, will be unfamiliar. They represent the case where there is

a common "don't know" or neutral value (0), and a number of extreme positions, the *metae*[11], with perhaps some intermediate values.

I have been trying to write an Expert System for Statistics, which would elicit a description of an experiment from a client who is ignorant of Statistics, and automatically work out a valid (though very likely suboptimal) way of analysing the data when and if they become available. Amongst other benefits, this has forced me to look hard at how measurements are described. The common classification into nominal, partially ordered, ordinal, ordered metric, interval, and ratio, will almost certainly be unfamiliar to a naive client. This turns out to be no great loss, as it is inadequate to support automatic method selection. (Such distinctions as "ordinal but with an underlying continuous scale" and "strictly positive" are also needed.) More precise descriptions of measurement scales are not only needed by an Expert System, but are easier for the client to provide.

One of the questions I had to consider was dichotomising an ordinal variable. In general, the best we can do is to dichotomise around a (possibly pooled) median, but some ordinal scales come with a built in zero. I have found three kinds of ordinal scale with a zero:

1. Scales resulting from the comparison of two objects or sensations.

2. Approximations to ratio scales (often differences).

3. Others.

In the first two kinds of scale, there is a single property being tested, and when one of the objects has (or is judged by the subject to have) more of "it" the outcome is "positive", and when it has less the outcome is "negative".

But we often see questions like this: "Do you vote for

1. the Conservative Party all the time

2. the Conservative Party some of the time

3. neither party

4. the Labour Party some of the time

5. the Labour Party all the time ?"

This is a multimetic scale, with two major parties and a number of small ones omitted to produce a bimetic scale. We very often see questions like this analysed as if the answers formed a simple ordinal scale.

Another example is "Are you

1. Strongly in favour of Capital Punishment

---

[11]The pyramidal columns at the end of the Roman circus, hence the triangular shapes

2. Moderately in favour of Capital Punishment

3. No opinion

4. Moderately opposed to Capital Punishment

5. Strongly opposed to Capital Punishment?"

At first sight this looks like a straightforward comparison, with the two objects being two opinions and the property being compared being the degree to which the subject holds each. But is it really the case that the same phenomenon explains subject A's strong opposition rather than moderate opposition and subject B's uncertainty rather than moderate support? Indeed, can we not have a subject who says "I think murderers deserve to die, but that the State has no right to kill them" and so wants to tick two boxes?

B.3. Dealing with Multimeties.

It is best to avoid them in the first place by having a separate question for each goal. This of course leads to extra work in the survey design and in its analysis, but the reward is more meaningful data.

An easy way of coping is to collapse the scale to a simple nominal scale, with one value for the zero point and one for each of the extremes, with intermediate stages mapped to either the zero or the appropriate extreme as seems fit to the investigator. This is like dichotomising. If a scale is truly multimetic, it seldom makes sense to group the zero point with any of the other cases.

My limited experience suggests to me that bimeties are always the result of poor design, and that disentangling the metae into separate scales always gets closer to what the client really wants to know. Others with more experience may know of genuine uses for multimeties, and if so, we should develop methods for them.

Suppose we have a sample of N subjects, with an interval variable Y_i and a multimetic variable X_i measured on each subject. Suppose X has M metae, and meta j has M_j stages. (Thus in figure 5 M = 4 and M_j = 2 for j=1...4.) We define $M_0 = 0$. After regrouping Y by the values of X, an attractive model is

$$Y_{jkl} = mu + theta_{jk}.alpha_j + epsilon_{jkl}$$
$$\text{for } j = 1...M,\ k = 1...M_j$$

where

$$0 \le theta_{j1} \le \ldots \le theta_{jM_j} = 1$$
$$epsilon_{jkl} \text{ are i.i.d. with } E[epsilon_{jkl}] = 0$$

Solving this by least squares leads to a nonlinear problem, but it is reasonably well behaved, and good starting values for mu and $alpha_j$ are available from the group means.